AD-A076 874 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 9/2

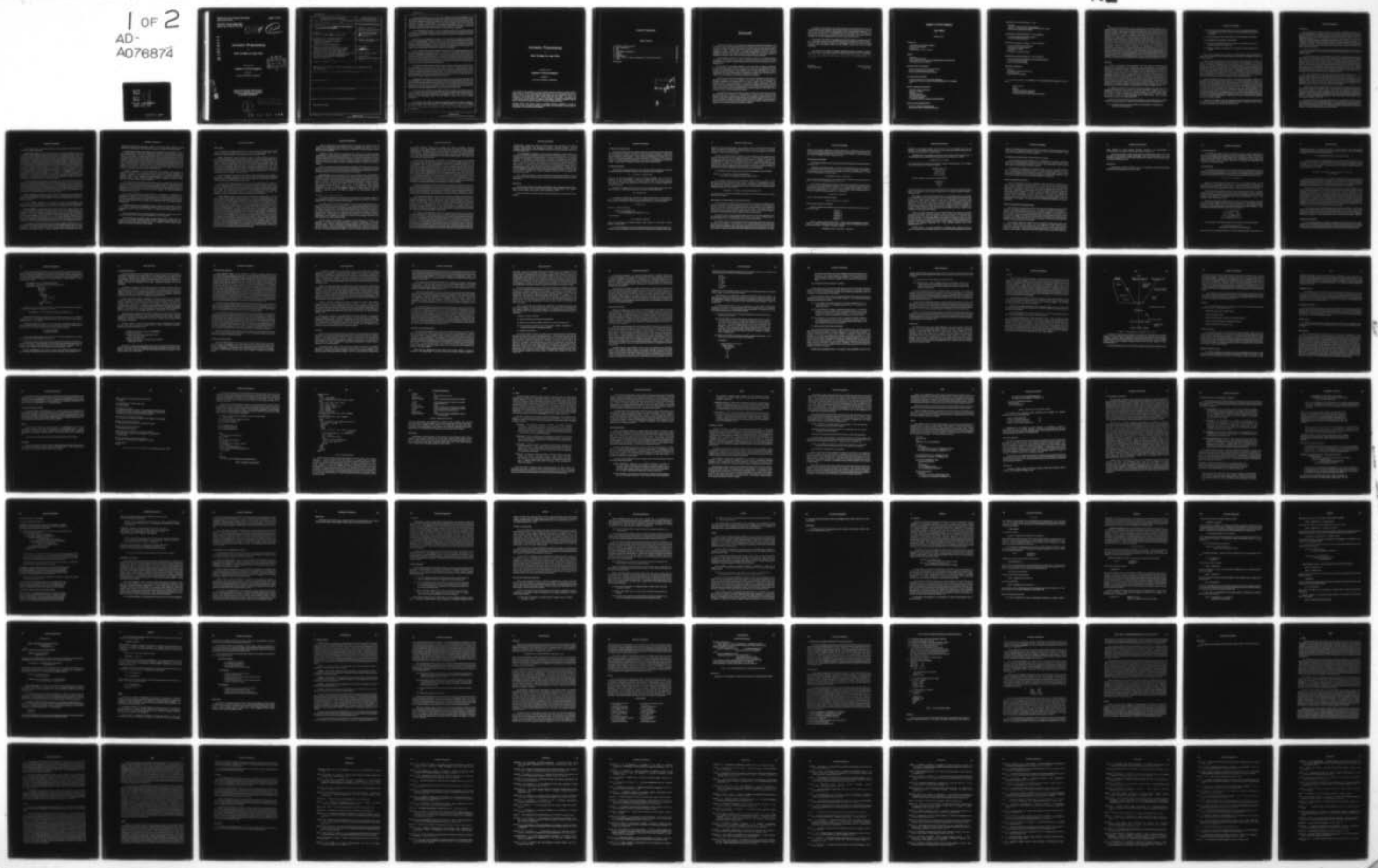
AUTOMATIC PROGRAMMING.(U)

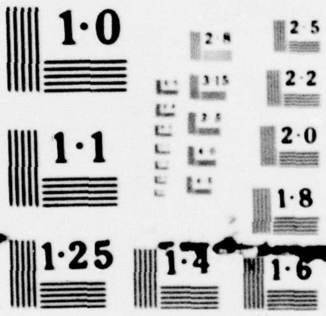
AUG 79 R ELSCHLAGER , J PHILLIPS MDA903-77-C-0322

UNCLASSIFIED STAN-CS-79-758 NL

1 OF 2

AD-
A076874




1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
1·8
1·25
1·4
1·6

Stanford Heuristic Programming Project
Memo HPP-79-24

August 1979

Computer Science Department
Report No. STAN-CS-79-758

12

Automatic Programming

by

Robert Elschlager and Jorge Phillips

a section of the

Handbook of Artificial Intelligence

edited by

Avron Barr and Edward A. Feigenbaum

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	9 READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 HPP-79-24 (STAN-CS-79-758, HPP-79-24) / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER Technical rept.
4. TITLE (and Subtitle) 6 Automatic Programming (~~a section of the Handbook of Artificial Intelligence~~).	5. TYPE OF REPORT & PERIOD COVERED technical, August 1979
	6. PERFORMING ORG. REPORT NUMBER HPP-79-24 (STAN-CS-79-758)
7. AUTHOR(s) 10 Robert/Elschlager ~~and~~ Jorge/Phillips (A. Barr and E.A. Feigenbaum, editors)	8. CONTRACT OR GRANT NUMBER(s) 15 ~~ARPA~~ MDA 903-77-C-0322, PHS NIH-RR-00785-06
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University Stanford, California 94305 USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 12 98
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Ave., Arlington, VA 22209	12. REPORT DATE 11 Aug~~ust~~ 1979
	13. NUMBER OF PAGES 97
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Mr. Philip Surra, Resident Representative Office of Naval Research, Durand 165 Stanford University	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see reverse side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 094120

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Those of us involved in the creation of the Handbook of Artificial Intelligence, both writers and editors, have attempted to make the concepts, methods, tools, and main results of artificial intelligence research accessible to a broad scientific and engineering audience. Currently, AI work is familiar mainly to its practicing specialists and other interested computer scientists. Yet the field is of growing interdisciplinary interest and practical importance. With this book we are trying to build bridges that are easily crossed by engineers, scientists in other fields, and our own computer science colleagues.

In the Handbook we intend to cover the breadth and depth of AI, presenting general overviews of the scientific issues, as well as detailed discussions of particular techniques and important AI systems. Throughout we have tried to keep in mind the reader who is not a specialist in AI.

As the cost of computation continues to fall, new areas of computer applications become potentially viable. For many of these areas, there do not exist mathematical "cores" to structure calculational use of the computer. Such areas will inevitably be served by symbolic models and symbolic inference techniques. Yet those who understand symbolic computation have been speaking largely to themselves for twenty years. We feel that it is urgent for AI to "go public" in the manner intended by the Handbook.

Several other writers have recognized a need for more widespread knowledge of AI and have attempted to help fill the vacuum. Lay reviews, in particular Margaret Boden's Artificial Intelligence and Natural Man, have tried to explain what is important and interesting about AI, and how research in AI progresses through our programs. In addition, there are a few textbooks that attempt to present a more detailed view of selected areas of AI, for the serious student of computer science. But no textbook can hope to describe all of the sub-areas, to present brief explanations of the important ideas and techniques, and to review the forty or fifty most important AI systems.

The Handbook contains several different types of articles. Key AI ideas and techniques are described in core articles (e.g., basic concepts in heuristic search, semantic nets). Important individual AI programs (e.g., SHRDLU) are described in separate articles that indicate, among other things, the designer's goal, the techniques employed, and the reasons why the program is important. Overview articles discuss the problems and approaches in each major area. The overview articles should be particularly useful to those who seek a summary of the underlying issues that motivate AI research.

Eventually the Handbook will contain approximately two hundred articles. We hope that the appearance of this material will stimulate interaction and cooperation with other AI research sites. We look forward to being advised of errors of omission and commission. For a field as fast moving as AI, it is important that its practitioners alert us to important developments, so that future editions will reflect this new material. We intend that the Handbook of Artificial Intelligence be a living and changing reference work.

The articles in this edition of the Handbook were written primarily by graduate students in AI at Stanford University, with assistance from graduate students and AI professionals at other institutions. We wish particularly to acknowledge the help from those at Rutgers University, SRI International, Xerox Palo Alto Research Center, MIT, and the RAND Corporation.

The authors of this chapter on Automatic Programming research are Robert Elschlager and Jorge Phillips. They have worked from material supplied by the AP researchers themselves, including David Barstow, Cordell Green, Neil Goldman, George Heidorn, Elaine Kant, Zohar Manna, Brian McCune, Gregory Ruth, Richard Waldinger, and Richard Waters.

Automatic Programming

by

Robert Elschlager and Jorge Phillips

a section of the

Handbook of Artificial Intelligence

edited by

Avron Barr and Edward A. Feigenbaum

This research was supported by both the Defense Advanced Research Projects Agency (ARPA Order No. 3423, Contract No. MDA 903-77-C-0322) and the National Institutes of Health (Contract No. NIH RR-00785-06). Jorge Phillips' work was supported in part at the Stanford AI Lab (ARPA Order 2494, Contract MDA 903-76-C-0206). The views and conclusions of this document should not be interpreted as necessarily representing the official policies, either express or implied, of the Defense Advanced Research Projects Agency, the National Institutes of Health, or the United States Government.

Automatic Programming

Table of Contents

Foreword

Those of us involved in the creation of the Handbook of Artificial Intelligence, both writers and editors, have attempted to make the concepts, methods, tools, and main results of artificial intelligence research accessible to a broad scientific and engineering audience. Currently, AI work is familiar mainly to its practicing specialists and other interested computer scientists. Yet the field is of growing interdisciplinary interest and practical importance. With this book we are trying to build bridges that are easily crossed by engineers, scientists in other fields, and our own computer science colleagues.

In the Handbook we intend to cover the breadth and depth of AI, presenting general overviews of the scientific issues, as well as detailed discussions of particular techniques and important AI systems. Throughout we have tried to keep in mind the reader who is not a specialist in AI.

As the cost of computation continues to fall, new areas of computer applications become potentially viable. For many of these areas, there do not exist mathematical "cores" to structure calculational use of the computer. Such areas will inevitably be served by symbolic models and symbolic inference techniques. Yet those who understand symbolic computation have been speaking largely to themselves for twenty years. We feel that it is urgent for AI to "go public" in the manner intended by the Handbook.

Several other writers have recognized a need for more widespread knowledge of AI and have attempted to help fill the vacuum. Lay reviews, in particular Margaret Boden's Artificial Intelligence and Natural Man, have tried to explain what is important and interesting about AI, and how research in AI progresses through our programs. In addition, there are a few textbooks that attempt to present a more detailed view of selected areas of AI, for the serious student of computer science. But no textbook can hope to describe all of the sub-areas, to present brief explanations of the important ideas and techniques, and to review the forty or fifty most important AI systems.

The Handbook contains several different types of articles. Key AI ideas and techniques are described in core articles (e.g., basic concepts in heuristic search, semantic nets). Important individual AI programs (e.g., SHRDLU) are described in separate articles that indicate, among other things, the designer's goal, the techniques employed, and the reasons why the program is important. Overview articles discuss the problems and approaches in each major area. The overview articles should be particularly useful to those who seek a summary of the underlying issues that motivate AI research.

Eventually the Handbook will contain approximately two hundred articles. We hope that the appearance of this material will stimulate interaction and cooperation with other AI research sites. We look forward to being advised of errors of omission and commission. For a field as fast moving as AI, it is important that its practitioners alert us to important developments, so that future editions will reflect this new material. We intend that the Handbook of Artificial Intelligence be a living and changing reference work.

The articles in this edition of the Handbook were written primarily by graduate students in AI at Stanford University, with assistance from graduate students and AI professionals at other institutions. We wish particularly to acknowledge the help from those at Rutgers University, SRI International, Xerox Palo Alto Research Center, MIT, and the RAND Corporation.

The authors of this chapter on Automatic Programming research are Robert Elschlager and Jorge Phillips. They have worked from material supplied by the AP researchers themselves, including David Barstow, Cordell Green, Neil Goldman, George Heidorn, Elaine Kant, Zohar Manna, Brian McCune, Gregory Ruth, Richard Waldinger, and Richard Waters.

Avron Barr
Edward Feigenbaum

Stanford University
July, 1979

Handbook of Artificial Intelligence

Topic Outline

Volumes I and II

The following sections of the Handbook are still in preparation and will appear in the third volume:

Theorem Proving
Vision
Robotics
Information Processing Psychology
Learning and Inductive Inference
Planning and Related Problem-solving Techniques

Automatic Programming (AP) is a new, dynamic, and not precisely defined area of artificial intelligence. This overview discusses the definitions, history, motivating forces and goals of automatic programming and includes a brief description of the basic characteristics and central issues of AP systems. The article begins with a section discussing the various possible definitions of automatic programming, the background in which it has achieved existence, as well as some of its general motivating forces and goals. The next section describes four characteristics of all AP systems: the method by which a user of such a system specifies or describes the desired program, the target language in which the system writes the program, the problem or application area to which the system is addressed, and the approach or operational method employed by the system. Next, a section discusses four basic issues, one or more of which concern all AP systems: the representation and processing of partial or incomplete information; the transformation of structures, and especially the transformation of program descriptions into other descriptions (in this chapter, the term program description includes the user's specification of the desired program, any internal representations of the program, as well as the target language implementation); the efficiency of the target language implementation; and the system's capabilities for aiding in the understanding of the program. Following this overview, the reader will find articles on the methods of specifying programs in AP systems, on some of the basic operational methods employed in such systems, and then eight articles describing most of the major AP projects.

Definition

The bulk of the research in AP has appeared in the 1970s, and it is not surprising that there is lack of agreement as to the definition, scope, and direction of the endeavor. Several brief definitions of automatic programming have been suggested in the literature, but considering the newness of the area, one should not expect these definitions to be precise. One definition says simply that AP is something that will save people the chores of programming (Biermann, 1976a). Another states that an AP system carries out part of the programming activity currently performed by a human in constructing, a program written in some machine executable language, given the definition of the problem to be solved; here, the essence of an AP system is that it assumes some responsibilities otherwise borne by a human, and thereby reduces the person's task (Hammer & Ruth, 1979). Yet another states that AP means having the computer help write its own programs (Heidorn, 1977). AP is the application of a computing system to the problem of effectively utilizing that or another computing system in the performance of a task specified by the user (Balzer, 1973b).

To summarize, perhaps we can define AP here as an automation of some part of the program-writing activities that currently are typically performed by people and not yet performed by machine. Therefore the definition excludes such systems and software environments as assembly languages and high-level languages such as FORTRAN, COBOL, PL/1, ALGOL, or LISP; and such programming aids as symbol tables, cross reference generators, text editors, and debugging systems.

Other more extensive definitions have been suggested. One definition (Balzer, 1973b) "rates" AP systems according to a measure of merit, which includes the following factors:

(a) the amount of time and effort needed by the programmer to formulate and specify the desired program;

(b) the efficiency of the decisions made by the system in designing the program, and consequently the overall efficiency of the program that is produced by the system;

(c) the ease with which future modifications can be incorporated in the program;

(d) the reliability and ruggedness of the program;

(e) the amount of computer resources, including time and memory, used by the system to produce that program; and

(f) the range, as well as the complexity, of the tasks that can be handled by the system.

Notice that, according to such a measure, a FORTRAN language compiler would rank as an AP system. However, its rank would be significantly less than the potential of current AP research projects.

Another source (see article D3) lists some specific factors that bear on factor (a) above, the factor concerned with the effort required of the programmer. The specific factors are informality, language level, and executability. An AP system is informal to the degree that the user can be ambiguous (various interpretations of the specification are possible) and partial or incomplete (pieces of information, including perhaps information about referencing and sequencing, have been omitted). Language level refers to the degree to which the AP system can accept specifications in a terminology natural to the problem area under consideration. Executability refers to the degree to which the system can achieve a desired program state on the basis of a description of that state, that is, the degree to which the user need only specify what is wanted rather that how to obtain it.

Another definition of AP can be obtained by defining the development phases of a software system (software development refers to the creation of a program or collection of programs, from their inception to the completed product). On this basis, it would follow that AP assists the programmer with one or more of these phases. For example, in a later article that describes the PROTOSYSTEM research project D7, the development of data-processing systems (programs) is seen as passing through five phases. First, the programming problem is defined by clearly identifying and understanding what the desired software is to accomplish; second, what the program is to do in order to alleviate this problem is clearly and precisely determined; third, the organization, flow of control, and data representations are selected from standard implementation possibilities; fourth, this very high-level specification in terms of standard implementations is transformed into code in some high-level language; and fifth, this code is compiled.

These, then, are some of the more detailed definitions that have been presented for AP. Altogether, they define a somewhat amorphous direction of research; there is still no widespread agreement as to exactly what constitutes AP.

Background

The present period is not the first time the term automatic programming has been used. The term was employed once before, about twenty years ago, to mean writing a program in a high-level language (e.g., FORTRAN) and having a compiler transform the program into machine language code. Thus, one finds "Automatic Coding," Franklin Institute, January 1957 (see **Automatic Coding, 1957**), or *The Annual Review of Automatic Programming*, first appearing in 1959 (see **The Annual Review in Automatic Programming, 1960**). At that time, when "real" programming referred to writing a program in machine or assembly language, AP meant writing a program in FORTRAN. Today, when most programming is done in high-level languages, AP means programming in a software environment much more advanced than the ones created by these high-level languages.

Though the early meaning of the term automatic programming differs from the current meaning, nevertheless, at both times AP meant assisting and automating the process of writing programs.

In a general way, the forces responsible for AP twenty years ago are similar to those responsible for its appearance today. At both times there was a feeling that programmers were burdened with the need to specify many details, with the need to keep track of the many relations between these details, and with a programming environment that was not, perhaps, natural to the way in which they thought about the problem. At both times there was a feeling among some that new programming environments might be within grasp (twenty years ago the new environments were high-level languages) and that the software technologies required to realize such environments might be feasible. Out of the desire for new programming environments and out of the feeling that these new environments might be attainable, there appeared, in each period, an endeavor called AP.

The current motivations for AP, while similar to those twenty years ago, are more intense. Today software is costly and unreliable. Much time, money, and effort is currently being expended, with even greater expenditures forecast for the future. Software is seldom produced within budget or on time. Quite often the supposedly finished product, when delivered, fails to meet specifications. As programming applications of increasingly greater complexity are addressed, not only does reliability become more difficult to attain, but the costs of software, in terms of time, money, and effort, spiral upward.

To help alleviate these problems, AP aims at a general goal: To restyle the way in which the programmer specifies the desired program. This restyling should allow the programmer to think of the problem at a higher and more natural level. AP would like to relieve the programmer of mundane portions of programming, that is, the need to keep track of inordinate amounts of details. By changing the programming environment, AP could allow programmers to construct, with greater ease and with greater accuracy, the programs of the present and the more complex programs of the future.

This goal circles back to a succinct definition of AP: The computer is used as a tool that automates part of the programming process. That is, the computer performs a portion of the program-writing activities. Neither the goal nor this definition are especially precise, but the next sections are more specific. They describe the common characteristics and primary issues of AP systems..mark Characteristics of AP Systems

All AP systems have a specification method, a target language, a problem area, and an approach or method of operation.

Specification method Users of an AP system must be given some means or method for conveying to the system the program that they desire. This means is referred to as the *specification method* of the AP system. As will be seen in the remainder of this chapter, AP systems possess a variety of specification methods. Formal specification methods are those that might be considered to be *very high-level* programming languages. In general, the syntax and semantics of such methods are precisely and definitely defined. Formal methods also tend to be *complete*; that is, the specification will completely and precisely indicate what it is that the program is to accomplish, though, of course, the specification may not indicate the form of the program or how the program is to accomplish it. On the one hand, many formal specification methods are not usually *interactive*, which is to say, the system does not interact with the user in order to obtain missing information, to verify hypotheses, or to point out inconsistencies in the specification. For example, it is comparable to the passive acceptance of a program's specification by a compiler of a high-level language (e.g., FORTRAN). On the other hand, there are some formal specification methods that are interactive (see McCune, 1978, which emphasizes interactive formal specification techniques as a natural extension of incremental compiling).

A different method of specification is by examples. Here the user specifies the desired program by simply giving examples of what the desired program is to do; the AP system would then construct the desired program. The specification might consist of examples of the input/output behavior of the desired program, or it might consist of traces of the desired program's behavior (a trace is an example showing how the program should process a given input). Specification by examples (or traces) is certainly not complete: The examples do not fully describe in all cases the behavior of the desired program.

Natural language (e.g., English) is another method of specification. The user specifies in natural language what the desired program is to do. This method is often interactive (cf. articles on PSI and NGPS), checking hypotheses, pointing out inconsistencies, and asking for further information.

A more detailed discussion of specification, including some advantages and disadvantages of the various methods, is presented in the article on program specification. Examples of program specification are found in most of the remaining articles of this chapter.

Target language The specification method refers to the input to the AP system, and the target language is concerned with the system's output of the finished program. The language in which the AP writes the finished program, or parts of the finished program, is called the *target language*. The target languages of the AP systems described in this chapter are high-level languages such as LISP, PL/1, or GPSS. As an example, suppose that the target language of an AP system were LISP. The user, possibly employing a very high-level language, or examples, or natural language, would specify to the AP system what the desired program is to do. Then the AP system would eventually output a LISP program to do just that.

It is possible to view specification method and target language as relative terms. In an AP system that carries the process of writing programs through several phases, the input language for each phase could be thought of as a specification method, and the output specification as being written in a target language, which then becomes the input specification method to the next phase. However, in this chapter, target language is usually reserved for the language in which the output program of the whole AP system is written.

Problem area Another characteristic of an AP system is its problem area or area of intended application. Problem area, problem domain, application area, and application domain are synonomous terms. For some AP systems, the scope of its problem area is relatively precise; for example, the problem area of the NLPQ system is simple queuing problems, while the problem area of the PROTOSYSTEM project is input/output intensive data-processing systems, including inventory control, payroll, and other record-keeping systems. On the other hand, the problem area of some AP systems can be relatively large; the application domain of the PSI system is symbolic computation, including list processing, searching and sorting, data storage and retrieval, and concept formation. The problem area of a system can have a bearing on the method of specification, introducing relevant terminology, influencing the method of operation or approach used by the AP system, etc.

Method of operation The fourth characteristic of AP systems is the approach or method of operation. AP is too new for there to be very many clear-cut categories of methods of operation. The approach(es) of most systems is not easily categorized. A separate article on basic approaches discusses some of the more clear-cut methods, including theorem proving, program formation, knowledge engineering, automatic data selection, traditional problem solving, and induction.

In the theorem-proving approach, the user specifies the conditions that must hold for the input data (to the desired program) and the conditions that the output data should satisfy. The conditions are specified in some formal language, often the predicate calculus. A theorem prover is then asked to prove that, for all given inputs, there exists an output that satisfies the output conditions. The proof, then, yields a program. The desired program can be extracted from the proof.

The program transformation approach refers to transforming a specification or description of a program into an equivalent description of the program. The reason for the transformation might be to convert a specification that can be easily written and read into one that is more complicated but more efficient; alternatively, the goal might be to convert a very high-level description of the program into a description closer to a target language implementation.

Knowledge engineering (see Applications chapter), applicable to many areas in addition to AP, refers to identifying and explicating knowledge; and it often means "realizing" the knowledge as specific rules that can be added to or removed from the *knowledge base* of a system.

Traditional problem solving (see section Search), also applicable to many areas, refers to the use of goals to direct the choice and application of a set of operators.

These approaches or paradigms overlap, and many systems utilize a method that may, in part, draw on elements from several. While it is hard to categorize the approaches of AP systems, there are now enough systems so that it is possible to identify some common issues, and these are the topic of the next section.

Basic Issues

In the article on basic approaches and in all the articles describing the individual research projects, the reader will find one or more of several explicit basic issues addressed: partial information, transformation, efficiency, and understanding.

Partial Information Partial information pertains to systems whose methods of specification allow for partial or fragmentary descriptions of the desired program: Not all of the required information is present in the specification, or, where it is present, it may not be explicit. Since the problem of partial information does not apply to systems that have complete methods of specification, systems such as DEDALUS, PROTOSYSTEM I, LIBRA, and PECOS are not concerned with this problem. On the other hand, systems that accept incomplete specifications, especially natural language specifications, are very much concerned with partial information. The NLPQ, PSI, and SAFE systems fall in this category. A classification of the different kinds of missing information that might occur in a natural language specification is given in the SAFE article.

Usually going hand in hand with the problem of partial information is the problem of consistency. Incomplete methods of specification often permit inconsistency between different parts of the same specification. In such cases, the system must check for inconsistencies and, if they are found, resolve them.

In trying to fill in missing information in one part of the specification or checking for consistency between different parts and resolving any discovered inconsistency, the system may use information that occurs either explicitly or implicitly in other parts of the specification. Also, it might utilize a knowledge base containing information about the problem area. Finally, the system may consult the user in an attempt to gain the sought-for information. One of the explicit devices for utilizing such information is *constraints*. For examples of these, see the article on PSI and especially the article on SAFE.

Transformation Another issue addressed by AP systems is transformation. The term refers, simply, to transforming a program description, or part of a program description, into another form. All AP systems use transformation, if only to transform an internal description of the program into a target language implementation (description). Even a compiler of high-level languages (e.g., FORTRAN, PL/I, ALGOL) will often transform a program description several times, taking it through several internal representations, the last of which is the machine language description. However, a compiler differs from an AP system in that it applies the transformations in a rigid, predetermined manner; in an automatic programming system there might be no predetermined way to apply the transformations, the application depending on an analysis and exploration or the results of applying various transformations. Systems, such as DEDALUS and PECOS, that use extensive transformation on the program description have a knowledge base containing many transformation rules that convert parts of a higher level description into a lower level description, closer to a target language implementation. Such rules are repeatedly applied to parts of the program description with the goal of eventually producing descriptions within the target language. These systems develop a tree of possible descriptions of the program, with each descendant of a node being the result of a transformation. One of the goals, then, in developing the tree is to find a description that is a target language implementation of the desired program. Another goal might be to find an efficient target language implementation.

Other AP systems may use transformation rules in various ways. For instance, the NLPQ system uses transformation rules to parse the natural language input from the user, to generate natural language output to the user, and to generate the target language program from an internal description.

Efficiency Another concern of AP systems is the efficiency of the target language implementation. The two projects that dealt with this issue are PROTOSYSTEM I and the PSI subsystem LIBRA. While the PROTOSYSTEM approach to creating efficient programs combines artificial intelligence with the mathematical technique of dynamic programming, the LIBRA approach uses a more extensive range of artificial intelligence techniques, employing a variety of heuristics, estimates, and kinds of knowledge to guide its search for an efficient program.

When it is said that an AP system optimizes a program for efficiency, it does not mean that the system finds the absolutely most efficient implementation; combinatorial explosion makes such a task impossible. Instead, optimizing means making some reasonable choices in the implementation so as to achieve a reasonably efficient program.

Understanding The basic concern of one of the systems below, PROGRAMMER'S APPRENTICE, pertain more to "understanding" the program than it does to the basic concerns of partial information, transformation, or efficiency. In this situation, understanding a program might be defined as that which enables a system to talk about, analyze, modify, or write parts of a program. It is the intention of the PROGRAMMER'S APPRENTICE, though it should be kept in mind that at present this system is not yet operational, to realize program understanding through the explicit use of *plans*. A plan represents one particular understanding or way of viewing a program, or part of a program (for a more detailed explanation, see the article on PROGRAMMER'S APPRENTICE). Understanding in the other systems is relatively implicit and does not reside in any one particular class of structures.

Overview of the Systems Articles

The projects described in the system articles cover much of the current research in AP, including the four basic issues just discussed: transformation rules, search for efficiency, handling partial information, and explicit understanding.

The NLPQ system is the first AP system to utilize natural language dialogue as a specification method. The user specifies part of a simple queuing simulation problem in English, and then the system, as is necessary, answers questions posed by the user, as well as queries the user in order to complete missing information or to resolve inconsistencies. The partial knowledge that the system has obtained about the desired program is represented as a semantic net that is eventually used to generate the program in the target language GPSS. Transformation or production rules analyze the user's natural language specification, build and modify the semantic net, produce natural language responses, and finally generate the target language program.

The PSI system is more recent and consists of many subsystems; it stresses the integration of a number of different processes and sources of knowledge. The problem application area is symbolic programming, including information retrieval, simple sorts, and concept formation. The user can specify the desired program with a mixture of examples and mixed-initiative natural language dialogue; for an easier and more natural interaction with the user, the system maintains and utilizes a tree of the topics that occur during the specification dialogue. Through such a dialogue, PSI creates a complete, consistent description of the desired program. In the last phase, the system explores repeated application of transformation rules in order to convert the description into a target language implementation. This last phase, the synthesis phase, is carried out by two subsystems: PECOS provides suitable transformation rules and LIBRA directs and explores the application of the rules, with the goal of obtaining an efficient target implementation. PECOS and LIBRA are described in separate articles.

Both PECOS and DEDALUS are examples of full-fledged, dynamic transformation systems. They each start out with a complete specification of the desired program. Each has a knowledge base of many transformation rules that are repeatedly applied to the specification. These repeated applications produce a sequence of specifications that eventually terminate with a specification that is a target language implementation. Because more than one transformation rule can apply in some cases, each system actually develops a tree of specifications (descriptions), with eventually one or more of the final nodes of the tree being a program implementation within the target language. Part of the differences between these two systems lies in the fact that DEDALUS is concerned with the logic of such programming concepts as recursion and subroutine. On the other hand, PECOS is more concerned with the multiplicity of implementations of very high-level programming constructs and operations, because that is its task within the PSI system. Though PECOS stresses knowledge of various implementations and DEDALUS stresses knowledge of programming constructs, both are systems where transformation is the primary emphasis.

The SAFE system article contains an extensive description of constraints and their use in handling partial information. SAFE processes a variety of different kinds of constraints, in order to fill in different kinds of information in the specification of the desired program, and employs different methods of processing these constraints. There are constraints related to type of object referenced in the specification, as well as related to sequencing of steps. Constraints are processed by backtracking and by carrying out a form of symbolic execution.

One of the ideas of the SAFE project is that a completely specified program satisfies a very large number of constraints. Information in the user's partial, fragmentary specification (partial and fragmentary since the specification does not mention all objects explicitly, or partially mentions other objects and may not contain explicit sequencing of actions) combined with the many constraints that a formal program satisfies (and possibly with information from a knowledge base of the application area or, in special cases, from information obtained from queries to the user), taken together, fully determine a complete and formal description of the program. No other system deals in so central a way with partial information and constraints as does the SAFE system.

The LIBRA and PROTOSYSTEM I projects are concerned with efficiency of the target language implementation. LIBRA uses an artificial intelligence approach, while PROTOSYSTEM I uses a combination of some artificial intelligence with primarily the mathematical approach of dynamic programming. Dynamic programming, modified by approximations and heuristics, produces an optimized target language implementation. On the other hand, LIBRA guides the application of the transformation rules furnished by the PECOS subsystem of PSI and directs the growth of the resulting tree (see above discussion of PECOS) with the goal of finding an efficient target implementation. LIBRA determines and utilizes estimates of what it is likely to achieve by exploring the development of a particular node. LIBRA has knowledge about how its own allocation of space and time should influence its strategy in searching for an efficient implementation. Though both LIBRA and PROTOSYSTEM I are concerned with producing efficient implementations, they approach the problem in different contexts. The first explores configurations of a data-processing program and the second explores applications of transformation rules.

The PROGRAMMER'S APPRENTICE is not necessarily intended to write the program, but instead to function as an apprentice to the user, with the user writing none, some, or all of the program and the apprentice assisting with such tasks as writing parts of the program, checking for consistency, explaining pieces of program, and helping the user modify programs. The central concern of this project is *understanding*, through the explicit device of *plans*. A plan may be thought of as a template that expresses a viewpoint. Matching the plan to a part of a program description corresponds to understanding the part in that way. Several plans can match the same part of a program, corresponding to different ways of understanding that part. Plans can also be built up in a hierarchical fashion. The goal is that the PROGRAMMER'S APPRENTICE, with the understanding attained through the use of plans, can assist the programmer with correcting mistakes, writing parts of the program, and effecting modifications.

All of these are research projects: At present none has been responsible for an AP production system. Much research remains before most of these systems can be of use to programmers.

References

See The Annual Review in Automatic Programming (1960), Automatic Coding (1957), Balzer (1973a), Balzer (1973b), Balzer (1973c), Biermann (1976a), Hammer (1977), Hammer & Ruth (1979), Heidorn (1976), Heidorn (1977), and McCune (1978).

Further references for specific research areas are listed with the other articles in this chapter.

A. Methods of Specification

There must be some means or method by which the user conveys to the AP system the kind of program that he wants. This method is called the program specification. It might entail fully specifying the program in some formal programming language or possibly just specifying certain properties of the program. It might involve giving examples of the input and the output of the desired program, giving formal constraints on the program in the predicate calculus, or giving interactive descriptions of the program at increasing levels of detail in English. (Specification is introduced in general terms in the overview article.)

Formal Specifications

One method of formal specification is that used with the basic approach of theorem proving (see below for this basic approach). Here one might specify a program as

$$(1) \quad \forall\, s1\ (P(s1) \supset \exists\, s2\ Q(s1,s2))$$

where s1 are the input variables, and s2 are the output variables. P(s1) is the input predicate (or input specification); it gives the conditions that the inputs, s1, can be expected to satisfy at the beginning of program execution. Q(s2) is the output predicate (specification); it gives the conditions that the outputs, s2, of the desired program are expected to satisfy.

Expression (1) states that for all s1, the truth of P implies there is an s2 such that Q(s1,s2) is true. If there are no restrictions on the inputs, one may simply write

$$\forall\, s1\ \exists\, s2\ Q(s1,s2)\ .$$

For example, a program that computes the greatest common divisor of two integers x and y might be specified by taking P(x,y) as the condition that x and y are positive, and Q(x,y,z) as the condition that z is the greatest common divisor. P(x,y) could be written as

$$x > 0 \text{ and } y > 0\ ,$$

and Q(x,y,z) could be written as

$$\begin{array}{l} \text{divide}(z,x) \text{ and divide}(z,y) \text{ and} \\ \quad \forall r((r>0 \text{ and divide}(r,x) \text{ and divide}(r,y)) \supset z \geq r)\ . \end{array}$$

The expression

$$\forall\, x\ y\ \exists\, z\ (P(x,y) \supset Q(x,y,z))$$

would then state that for all positive integers x and y, there is a z such that z is their greatest common divisor.

In the basic approach for this kind of specification, the above expression is given to a theorem prover that produces a proof from which a program can be extracted (see basic approach of theorem proving below). One is required to give to the theorem prover enough facts concerning any predicates and functions that occur in P and Q so that (1) is provable. Thus, in the above, one would have to specify a number of facts concerning the predicates "divide", "<", and "≥" over the integers.

Another very similar method of specification is that used with the basic approaches of program transformation and of very high-level languages. This specification method stresses the use of entities that are not immediately implementable on a computer, or at least not implementable with some desired degree of efficiency. There is considerable leeway in this classification. For instance, in some program transformation systems the entities employed may be quite abstract, without any hint of the desired algorithm. In other systems the algorithm most naturally suggested by the specification of the program could be inefficient, but the AP system will produce an efficient but perhaps convoluted program.

One example of a specification used with program transformation is (see article D6)

```
gcd(x,y) ← compute max (z: divide(z,x) and divide(z,y))
           where x and y are nonnegative integers greater than zero .
```

This expression states that the gcd (greatest common divisor) of x and y is the maximum of all those *z* such that *z* divides x and y. Furthermore, it is assumed that x and y are nonnegative integers one of which is nonzero. By successive transformations of this definition of gcd, the system would produce an efficient recursive program. Another example (Darlington & Burstall, 1973, p. 280) is

```
factorial(x) := if x=0 then 1 else times(x,factorial(x-1)) .
```

The system, then, by various transformations produces a more efficient nonrecursive, though more tortuous, program.

Advantages and Disadvantages of Formal Specifications

The first specification method, that involving the input and the output predicates and based on formal logic, is completely general: Anything can be specified. On the other hand, the user must have a sufficient understanding of the desired behavior of the program in order to give a full formal description of the input and output. This understanding can sometimes be difficult, even for simple programs. Also, the present form of theorem provers and problem reduction methods makes synthesis of longer programs difficult.

The second type of formal specification does not have such arbitrary generality, but the terminology used in the specification often is closer to our way of thinking about a particular subject, and so it should be easier to create such specifications.

Even though some of the above formal methods are arbitrarily general and others are not, they all are complete: The specification of the desired program fully and completely specifies what the program is to do. This is not true of some of the other methods discussed below, where the specification does not uniquely determine what the program is to do. With such methods it becomes a concern whether the program produced by the system is actually what the user desires. Sometimes a system employing such a method may need to verify whether the program it produces is the program that the user wants. On the other hand, with the specification methods discussed here, there is no such problem. For further reading on this subject, see Sibel, Furbach, & Schreiber (1978).

Specification by Examples

Some simple programs are most easily described using examples of what the program is supposed to do.

Examples of input/output pairs In this specification method, the user gives examples of typical inputs and the corresponding outputs. Consider specifying or describing a concatenation of lists to someone who is unfamiliar with the term "concatenation." It might be most straightforward to use an example:

concat [(A B C), (D E)] = (A B C D E) ,

which states that when the input of the function "concat" consists of the two lists (A B C) and (D E), then the corresponding output is (A B C D E).

Given certain commonsense assumptions, this example input/output pair should suffice to specify what it is that the desired program is to do. In more complicated cases, where the commonsense assumptions are not sufficient, more examples must be given in order to specify the program uniquely. For instance, the above example could be misinterpreted as a "constant" program that always gave (A B C D E) as output:

concat [x,y] = (A B C D E) .

In such a case, giving an additional example

concat [(L M),(N O P)] = (L M N O P) ,

would probably clear up any confusion.

Another instance of this method is the specification of the function "prime" by a set of input/output pairs:

prime(1) = 1
prime(2) = 2
prime(3) = 3
prime(4) = 5
prime(5) = 7
prime(6) = 11

Generic examples of input/output pairs In certain cases, generalizations of specific examples or generic examples are more useful in order to avoid the problems inherent in partial specifications. For instance, the generic example

reverse [(X1 X2 X3 ... Xn)] = (Xn ... X3 X2 X1) describes a list reversal function. Here, the X1,X2,...,Xn are variables which may be anything. This specification is still partial but is more complete than any specification of this function given by example of input/output pairs.

Program traces Traces allow more imperative specifications than do example pairs. A sorting program may be specified with input/output pairs (e.g., Green et al., 1974):

```
sort [(3 1 4 2)] = (1 2 3 4) ,
```

but it would be hard to specify an insertion sort program in the same way. Yet, a program trace could express such a program as follows:

```
sort [(3 1 4 2)] --> ( )
       (1 4 2) --> (3)
         (4 2) --> (1 3)
           (2) --> (1 3 4)
            () --> (1 2 3 4)
```

Another example of specification by traces might be

```
gcd(12,18) ->
   (6,12) ->
    (0,6) ->
       6
```

for the specification by trace of the Euclidean algorithm that computes the greatest common divisor. An example of using a trace to specify part of a concept formation program is presented in D2.

More formally, a trace may be defined as follows. A programming domain can be thought of as consisting of a set of abstract objects, a set of possible representations (called *data structures*) for these abstract objects, a basic set of operators to transform these representations, and a class of questions or predicates that can be evaluated on these data structures. A *programming domain* thus characterizes a class of programs that might be constructed to operate on representations of the set of abstract objects in the domain. For a given program operating on some data objects in the domain, a *trace* is a sequence of changes of these data structures and control flow decisions that have caused these changes during execution of the program.

Traces are usually expressed in terms of domain operators and tests (or functional compositions of these). Traces are classified as *complete* if they carry all information about operators applied, data structures changed, control decisions taken, etc.; otherwise, they are called *incomplete*. An interesting subclass of the latter is the class of *protocols*, in which all data modifications are explicit but all control information (e.g., predicate evaluations that determine control flow) is omitted. A protocol is then a sequence of data structure state snapshots and operation applications (for a more complete definition see Artilce aplapproaches-???).

Generic traces Like generic examples of input/output pairs, these may also be useful. In general, there is a whole spectrum of trace specifications depending on how much imperative information and descriptive information is present in the trace. For instance, the trace above is completely descriptive; traces that contain function applications and/or sequencing information tend to be more imperative.

Advantages and Disadvantages of Specification by Examples

As stated above, generic examples are less ambiguous than non-generic examples. Traces are less ambiguous than input/output pairs, but the user is required to have in mind some idea of how the desired program is to function. On the other hand, traces do allow some imperative specification of the flow of control.

Specification by examples can be natural and easy for the user to formulate (Manna, 1977). Examples have the limitations inherent to informal program specifications: The user must choose examples so as to unambiguously specify the desired program. The AP system must be able to determine when the user's specification is consistent and complete and that the system's "model" of what the user wants is indeed the right program.

Natural Language Specifications

Given an appropriate conceptual vocabulary, English descriptions of algorithms are often the most natural method of specification. Part of the reason is that natural language allows greater flexibility in dealing with basic concepts than do, say, very high-level languages. This flexibility requires a fairly sophisticated representational structure for the model, with capabilities for representing the partial (incomplete) and often ambiguous descriptions that users provide. In addition, it may be necessary to maintain a database of domain-dependent knowledge for certain applications. Experience with implemented systems, such as SAFE (Balzer, Goldman, & Wile, 1977a; see also D3), suggests that the relevant issues are not in the area of natural language processing but in how the specifications are modeled in the system and what "programming knowledge" the system must have.

Mixed-Initiative Natural Language Dialogue

More versatile, this specification method involves interaction between the user and the system as the system builds and tries to fill in the details in its model of the algorithm. In addition to maintaining a model of the algorithm, such systems sometimes will even maintain a kind of model of the user to help the system tailor the dialogue to a particular user's idiosyncracies. Various techniques mentioned previously, such as examples or traces, could be used in the dialogue as a description of some part of the algorithm. The system might be designed so as to allow users to be as vague or ambiguous as they please; the system will ultimately ask them enough to fill in the model.

This method is probably the closest to the usual method of program specification used by people, allowing both the specifier and the programmer to make comments and suggestions. Users do not have to keep every detail in mind, nor do they have to present them in a certain order. The system will eventually question the user for missing details or ambiguous specifications. On the other hand, this method requires a system that deals with many problems of natural language translation, generation, and representation. A representation is also required for the system's model of the algorithm.

The PSI system (Green, 1976b; see also D2) and the NLPQ system (Heidorn, 1974; see also D8) use this method of program specification. *Floyd* (1972), and Green (1977), give hypothetical dialogues with such a system, illustrating the problems that researchers have encountered with this approach.

References

See Biermann (1976a) and Heidorn (1977). For examples of individual specification methods see the remaining articles of this chapter.

B. Basic Approaches

The following are some of the basic approaches used in Automatic Programming (AP) systems to synthesize desired programs from user specifications. There is not always a clear distinction between synthesis and specification. Furthermore, as will be seen from the later articles, some systems employ primarily one approach while others employ more elaborate paradigms that use several approaches. (Synthesis and specification are introduced in the overview article.)

Theorem Proving

The theorem-proving approach is used for the synthesis of programs whose input and output conditions can be specified in the formalism of the predicate calculus. As stated in the section on formal specifications, the user specifies the desired program for the theorem prover as an assertion to be proved. This assertion usually takes the form Green (1969):

$$\forall\, s1\ (\ P(s1) \supset \exists\, s2\ Q(s1,s2)\ )\ ,$$

where s1 is one or more input variables, s2 is one or more output variables, P is the predicate that s1 is expected to satisfy, and Q is the predicate that s2 is expected to satisfy after execution of the desired program. In addition to the above expression, the theorem prover must also be given enough axioms to make the above expression provable.

From the proof produced by the theorem prover, a program is *extracted*. For instance, certain constructs in the proof will produce conditional statements; others, sequential statements; and occurrences of induction axioms may produce loops or recursion. There are several variant methods of accomplishing these results (see Waldinger & Levitt, 1974, Kowalski, 1974, Clark & Sickel, 1977).

Although any interesting example would be far too long to work out in all of its detail here, it may be worthwhile to show how such a problem is set up. The interested reader is referred to Green, 1969, for a more complete development of the following example. Consider the very simple problem of sorting the dotted pair of two distinct numbers, in LISP. The axioms that would prove useful for this synthesis would be:

$$
\begin{aligned}
&1)\ x = \mathrm{car}\,(\mathrm{cons}(x,y))\\
&2)\ y = \mathrm{cdr}\,(\mathrm{c}[illegible](x,y))\\
&3)\ x = \mathrm{nil} \supset \mathrm{cond}(x,y,z) = z\\
&4)\ x \} \mathrm{nil} \supset \mathrm{cond}(x,y,z) = y\\
&5)\ \forall x,y\ (\mathrm{lessp}(x,y) \} \mathrm{nil} \leftrightarrow x < y)
\end{aligned}
$$

The specification of the desired program, and the theorem to be proved, would be:

$$
\begin{gathered}
\forall x.\ \exists y.\ [\mathrm{car}(x)<\mathrm{cdr}(x) \supset y=x] \wedge\\
[\mathrm{car}(x)\geq\mathrm{cdr}(x) \supset \mathrm{car}(x)=\mathrm{cdr}(y) \wedge \mathrm{cdr}(x)=\mathrm{car}(y)]\ ,
\end{gathered}
$$

which says that for every dotted pair input x, there is a dotted pair output y such that if x is already sorted, then y is the same as x; and if x is not sorted, then y is the interchange of the two elements of x. Using the techniques of resolution theorem proving (see **Theorem Proving.C**), we would obtain the following program:

```
y=cond(lessp(car(x),cdr(x)),x,cons(cdr(x),car(x))) .
```

In general, programs to be synthesized will not be as simple as the one above. One of the major problems that more complicated programs introduce is that they require some form of iteration or recursion for solution. To form a recursive program, one needs the proper induction axioms for the problem. A general schema for the induction axiom sufficient for most programs is Green (1969):

$$[P(h(\mathrm{nil}),\mathrm{nil}) \wedge \forall x[\mathrm{ATOM}(x) \wedge P(h(\mathrm{cdr}(x)),\mathrm{cdr}(x)) \supset P(h(x),x)]] \supset \forall z\,[P(h(z),z)]\ ,$$

where P is any predicate and h is any function. Somehow this predicate and function must be determined. Requiring the user to supply the induction axioms for each program to be synthesised somewhat defeats the purpose of the synthesis, yet having the system generate induction axioms until one of them works takes up far too much time and memory. Systems that determine the P and h usually use various heuristics to limit search.

There are several constraints inherent to the approach of theorem proving. First, for more complicated programs, it is often more difficult to correctly specify programs in the predicate calculus than it is to write the program itself. Second, the domain must be axiomatized completely, that is, one must give enough axioms to the theorem prover so that any statement that is true of the various functions and predicates that occur in the specification of the program can actually be proved from the axioms--otherwise, the theorem prover may fail to produce a proof, and thereby fail to produce the program. Third, present theorem provers lack the power to produce proofs for the specification of very complicated programs. To summarize, the user must fully and correctly specify the desired program, the theorem prover must be given enough axioms so that the specification is provable, and the theorem prover must be strong enough to prove the specification.

It should be noted that this approach does not allow partial specification: Users cannot specify the program partially, with the system helping them to fill in details. On the other hand, when a theorem prover does succeed in producing a proof of the specification, the correctness of the extracted program is guaranteed. Thus, AP systems might incorporate theorem proving where it is either convenient or where correctness is an important requisite.

Program Transformation

The transformation approach is used to automatically convert an easily written, easily understood LISP function into a more efficient, but perhaps convoluted program. One such system, described in Darlington & Burstall (1973), performs recursion removal, the elimination of redundant computation, expansion of procedure calls, and reuse of discarded list cells.

The recursion removal transforms a recursive program into an iterative one, which is generally more efficient, avoiding the overhead of the stacking mechanism. Candidates for recursion removal are determined by pattern matching the parts of the program against a recursive schema input pattern. If the match is successful and if certain preconditions are met, then the program is replaced by an iterative schema. A simple example of such a transformation rule is:

```
input pattern:  f(x) ::= if a then b else h(d,f(e));
precondition:  h is associative, x does not occur free in h;
result pattern: f(x) ::= if a
                     then result ← b
                     else begin
                        result ← d;
                        x ← e;
                        while not a
                           do begin
                              result ← h(result,d);
                              x ← e
                              end;
                        result ← h(result,b)
                        end
```

where a, b, d, e, f, and h in the input pattern are matched against arbitrary expressions in the candidate functions. For example, the function,

FACTORIAL(x) ::= if(x=1) then 1 else TIMES (x, FACTORIAL (x-1))

would match the above input pattern with f ← FACTORIAL, a ← (x=1), b ← 1, h ← TIMES, d ← x, and e ← (x-1). The resulting program would be the resulting pattern with these values substituted for a, b, d, e, f, and h.

Eliminating redundant computations includes traditional subexpression elimination as well as combining loops that iterate over the same range. The latter includes implicit iteration. Thus, if A, B, and C are represented as linked lists, the sequence:

X ← INTERSECTION (A,B)
Y ← INTERSECTION (A,C) ,

is really two implicit iterations, each over the set A. A suitable transformation rule would convert these into a single iteration over the set A.

Expanding procedure calls generally involves substituting the body of a procedure for each of the calls to it. The potential benefit arises from simplifications made possible by use of the ocal context. This technique is the starting point for a general class of transformations explored in Burstall & Darlington, 1975, and Wegbreit, 1975a.

Program transformation is also used to convert very high-level specifications into target language implementations (see D6, D5, as well as summaries of these articles in A).

Knowledge Engineering

AP systems are said to be "knowledge-based" when they are built by identifying and codifying the knowledge that is appropriate for the program synthesis and understanding (i.e., ability to manipulate and analyze programs) and by embedding this knowledge in some representation. Many of these systems use large amounts of many kinds of knowledge to analyze, modify, and debug large classes of problems. While the distinction is relative, it is possible to divide this knowledge into two types: programming knowledge and domain knowledge.

Programming knowledge includes both *programming language knowledge*, which is knowledge about the semantics of the target language in which the system will write the desired program, and *general programming knowledge*, which is knowledge about about such things as generators, tests, initialization, loops, sorting, searching, and hashing. Programming knowledge includes: (a) optimization techniques, (b) high-level programming constructs (loops, recursion, branching), and (c) strategy and planning techniques.

Domain knowledge is what is necessary for a system to infer how to go from the problem description or specification of a program in a certain program class (for example symbolic computation) to what needs to be done to solve the problem. This "know-how" includes how to structure the concepts in the domain or problem area and find interrelationships among them. It must also include knowledge about how to achieve certain results in the problem domain (cf., HACKER's learning of procedures Problem Solving.B5). Moreover, it should be able to define the problem in alternative ways and find alternative ways to solve the task--such knowledge represents an "understanding" of the domain.

Knowledge-based systems need a method of reasoning. Since they are not restricted to using the traditional formalisms of logic, they often supply their own flexible reasoning techniques for guiding the synthesis. Some of these techniques include inference, program simplification, illustration and simplification for the user, decision trees, problem-solving techniques, and refinement.

The basic concern in representing the knowledge is that the knowledge be structured in such a way that the search for relevant facts not cause a combinatorial explosion. Various representations employed include:

-- PLANNER-like procedural experts (AI Languages.C1),
-- Refinement rules (D5),
-- Modular, frame-like experts (OWL (Martin, 1974)
 and BEINGS (Lenat, 1975)),
-- Semantic nets (D8), and
-- Amorphous systems that try several ad hoc techniques
 ((Biggerstaff, 1976)).

Methods of accessing knowledge bases include: pattern invocation (Article D5), "when needed" (Sussman, 1975); frame relations and assertions, including filling in process models (Martin, 1974; Green, 1969; Lenat, 1975; see Articles D8, D2, and D3); and subgoal or case analysis (Green, 1977, and see D6).

Automatic Data Selection

This approach refers to the selection of efficient low-level data-structure implementations for a program specified in terms of high-level abstract information structures (e.g., sets). Generally, programming languages containing abstract data types have default representations that are a compromise between all likely uses of the structures; these data types are typically far from efficient in any one particular program. But a system with automatic data selection would choose, from a collection of possible implementations, an implementation more efficient for the particular program under consideration. For example, the abstract data type *set* could be represented in low-level implementations as a linked list, a binary tree, a hash table, a bit string, or as property list markings. Various operations on sets are easier in one representation than in another--e.g., set intersection using bit strings is simply a logical AND operation, while iteration over a set is easier when it is represented as a linked list--and some representations may not even be applicable in a given case (e.g., bit strings require that the domain of set elements be fixed and reasonably small, since one bit position is used for each possible element). Also, some representations may not permit all needed operations (e.g., the only way to enumerate the items in a set represented with property markings is to enumerate all atoms in the system.) By tailoring the representation to the particular programmer's intention, it is possible to produce much better code.

One such system performing data-structure selection for the user is Low, 1974, and Low, 1978. This system handles simple programs written in LEAP, a sublanguage of SAIL. It selects representations for sets, sequences, and relations from the fixed library of low-level data structures available in LEAP. The selection is guided by the goal of minimizing the product of the memory and time required to execute the resulting program.

The system begins with an information-gathering phase that searches out the relevant characteristics of the program's data structures, such as their expected size, number, the operations performed on them, and their interactions. Some of this information is obtained by questioning the user, and some is obtained by monitoring the actual execution of the program on typical data, using default representations for each structure. Then the system partitions into equivalence classes the variables whose values will be of the same type of data structure. The system employs a method similar to hill climbing (see Article Search.Overview) in order to determine a good assignment of data structures to the equivalence classes (i.e., the representations assigned to the equivalence classes are repeatedly varied, one at a time, to see if an improvement will result). For further details, see the above references.

Other AP systems are also concerned with the selection of an efficient set of data structures or file structures, but this concern is part of the general goal of writing an efficient program (see Articles D7 and D8).

Traditional Problem Solving

Traditional problem solving refers to using goals to direct the application of operations in a state space (see Search). The Heuristic Compiler (Simon, 1972) regards the task of writing a program as a problem-solving process using heuristic techniques, like those of GPS (see Article Search.D2). This pioneering work recognized the value of both a *state language*, to describe problem states and goals, and a *process language*, to represent the solver's actions.

In the Heuristic Compiler, the *State Description Compiler* is quite similar to later work on synthesis from examples. The program being synthesized is defined by specifying input/output conditions on the memory cells that it affects. The difference between the current state and the desired state is looked up in a table that specifies which operators to apply to transform the contents of the cells appropriately. The *Functional Description Compiler* is an important precursor to later work in automatic modification and debugging of programs. It uses a means-ends analysis to transform a known (compiled) routine into a new (desired) routine.

HACKER, a system described by Sussman (1975), adds to Simon's work, detecting and generalizing new differences (bugs) and defining appropriate operators to resolve them (patches). This system uses many significant AI techniques and language features: learning through practice how to write and debug programs; modular, pattern-invoked expert procedures (chunks of procedural knowledge); and hypothetical world models for subgoal analysis. Sussman's emphasis on generalizing from experience (trying old techniques in new situations), acceptance of the fact that users have an incomplete understanding of the desired program, and his goal-purpose annotation technique are all interesting directions in the development of Automatic Programming.

However, HACKER's preference for ruthless generation of "buggy" code without detailed planning has led to inadequate handling of subgoal conflicts. The user must carefully schedule the training sequences and be ready for the combinatorial explosion as the system exhaustively searches its base of world facts and programming knowledge. Such systems must constrain the search problem of large knowledge bases. Other attempts to distribute knowledge among interacting specialists have encountered the same difficulty (Lenat, 1975).

We find that systems such as HACKER, which have been designed to operate like human programmers, promise a moderate degree of success compared to knowledge-impoverished formal methods. However, these systems are still often hampered by the rigid formalism that governs their application: In what order are operators to be applied? How can domain-specific information be specified as differences? The formalisms used to incorporate the various knowledge sources in these systems seem too methodical; the method is space and time bound because it is based on search.

Induction

Induction or inductive inference refers to the system's "educated guess" at what the user wants on the basis of program specifications that only partially describe the program's behavior. Such specifications are often the examples of input/output pairs and program traces, in both regular and generic form (B). For each of these kinds of specification, the corresponding AP system must determine the general rules on the basis of a specification that contains only a few examples (or in the generic specifications, a limited class of examples) of the program behavior.

The work in *program synthesis from specification by examples* had its origin in research dealing with grammatical inference, where the objective was to infer a grammar that described a language, given several examples of strings of the language (Feldman, Gips, Horning, & Reder, 1969, and Biermann & Feldman, 1970). In a natural way, this research was associated with the inference of finite-state machines from the sequence (string) of states that the machine passes through during execution. The association was natural since finite-state machines are intimately related with the grammar that generates the strings of states that represent legal behavior of the machine (Biermann & Krishnaswamy, 1974). This research was the basis for two new avenues of investigation: synthesis from examples and synthesis from traces.

The crucial issue for program synthesis from examples is to develop a generalized program, that is, one that can account for more than the examples given in the program specification. To do this, these programs break down the input, looking for recursively solvable subparts (Shaw, Swartout, & Green, 1975) or computation repetitions that can be fitted into a known program scheme (Hardy, 1975).

The work in *program synthesis from trace specifications* seeks to invert the transformations observed in a trace protocol to create abstractions that generalize into loops and variables (Bauer, 1975). Of all the induction-based synthesis paradigms, it is the one that is closest to grammatical inference. Biermann & Krishnaswamy (1974) has built a system that interprets traces as directions through a developing flowchart. Phillips (1977) has implemented a system for the inference of very high-level program descriptions from a mixture of traces and example pairs in the context of a large automatic programming system D2.

All inductive inference systems are dependent upon a good *axiomatization of operations*. In other words, the system must know about all of the possible primitive operations that can be applied to the data structures if it is to hope to construct, by composition of these primitives, the desired program. Furthermore, a harmonious relation between the nature of the constructs in the specification and the most basic constructs in the target language is essential; for example, in Siklossy & Sykes, 1975, the tasks of tree traversal and repetitive robot maneuvers are directly translatable into LISP recursion. Moreover, these programs are required to know quite a bit about generalization. After synthesizing the program, they test it on other examples, sometimes by generating test cases and sometimes by asking the user for approval. For certain classes of programs, examples and traces provide a natural way for the user to specify what the desired program is to do.

Induction For Input/Output Pairs

The synthesis of programs from a specification consisting of instances of input/output pairs is strongly related to the problem domain to which these programs belong (e.g., sorting, concept formation). A set of program schemata characterize the entire class of programs for the domain. These schemata are like program skeletons and define the general structure of a program, omitting some details. The synthesis of a program thus amounts to (a) selecting a given schema that is representative of the program specified by the set of example pairs, and then (b) using the information present in the examples to instantiate the unfilled slots of the schema. So, there are two steps: a *classification* process, which selects the general structure (schema) of the target program, and an *instantiation* process, which completes the details of the target program.

What does the classification process require? Every schema defines a subclass of programs in the problem domain. Every set of example pairs defines a family of programs in the domain. Thus, the classification process must associate this set of example pairs with one of the subclasses of programs in the domain. In order to accomplish this task, a set of characteristics is associated with each schema (subclass) that, if present in the set of example pairs, guarantees that the set specifies a program of this type. Usually this task is accomplished by (a) providing a set of *difference measures* to be applied to the inputs and outputs of an example pair, as well as to different example pairs in the input collection (if it consists of more than one), and (b) providing a set of heuristics for each program schema that determine a *fit* measure of the example set that accompanies it. The task of classifying the example set is then simply reduced to choosing the schema with the highest fit value.

During the instantiation process, in addition to the difference fit measures described above, every schema has an associated set of rules for filling its empty slots through the extraction of necessary features from the examples. For instance, in the domain of list manipulation functions, cases where the output list contains all elements in the input and cases where the output list contains only every other element, etc., suggest different methods of constructing the output incrementally from the input. In the first case, the function maps down the input list; in the second case, it maps down the input using the LISP CDDR function. Slots are instantiated by these rules in terms of primitive operators of the domain and their functional compositions (in the above case, the basic LISP functions and their compositions).

Once a schema has been selected and instantiated, the synthesis algorithm must *validate* its hypothesis. This task is usually done either by generating some new examples for the program, evaluating the synthesized program on the example set, and checking the results with the user; or by presenting the program to the user and letting him/her verify its correctness.

In summary, the basic algorithm is:

(1) Apply the difference measures to the example set.

(2) Based on this application, classify the set into a particular schema class.

(3) Using heuristics associated with the particular schema, hypothesize a complete instantiation of the selected schema.

(4) Validate this hypothesis.

In this basic algorithm, if there is a single I/O pair in the specification, the difference measures are just a set of feature-detecting heuristics. If there is more than one pair, the pairs may be ordered according to the complexity of the input. Difference measures will fall into two classes: those that associate the structure of a pair with a schema class, and those that find differences between pairs. The latter are perhaps more crucial in the inference of a program. From these differences, a theory for the operation of the program is inductively inferred or, what is the same, a formation rule is derived. This operational theory might take the form of a certain schema class or of a recurrence equation that, in turn, specifies a schema class. In the classification phase it may be necessary to apply the classification rule to all pairs in order to infer the corresponding schema correctly. When several different schemas have been inferred, a decision rule is required to select the correct one.

An alternative approach is to reduce the whole problem to another paradigm for synthesizing programs. For example, if the problem domain has been formalized, so that there is a set of operators for the domain, it is possible to use a traditional problem solver to generate a solution to the input/output pair (considered as initial-state, goal-state) in the form of a sequence of operators that carry the input into the output. The solution so obtained can be considered a trace of the program to be synthesized and a trace-based paradigm may be employed.

Specification by examples is suitable for synthesizing a program only in those cases where the task domain is small and easily axiomatized. It may also be a feasible approach in the case where the domain is repetitious enough that a small set of pairs is sufficient to specify the program, which is almost never the case in practical programming domains. Such a specification method tends to be quite limited and does not lend itself to useful generalization to large domains. Nevertheless, the power of examples for clarifying concepts is unquestionable. It seems that the main application that this specification formalism will have in future automatic programming systems is restricted to the annotation and clarification of more formal program descriptions.

Induction From Traces

Inferring a program from a set of traces is, as mentioned earlier, very similar to inferring a description of a finite-state machine from a set of sequential states that the machine might pass through. The basic approach for synthesizing a program from a set of traces is to generate, in order of increasing complexity, the possible programs constructed from the programming-domain operators, tests, and their functional compositions; then, after each new program is generated, to validate the given traces against the program. If the generated program accounts for the traces, then it is the required solution. Notice that some kind of complexity measure is needed for the enumeration, for example program size (e.g., number of instructions in the program).

This basic approach suffers from the problems inherent to search in a large search space and thus admits improvements in the form of reduction of the combinatorial explosion by the use of heuristics to prune and guide the search process. It is thus not generally practical and is suited only to the inference of small programs in very simple domains. Nevertheless, it has been applied with moderate success to the inference of programs from memory traces. Usually consisting of register assignments, tests, and memory modification instructions, such programs and their traces are not very complex. Programs as complex as Hoare's FIND algorithm have been synthesized in this manner (Petry & Biermann, 1976). Though these systems tend to be knowledge-impoverished, Phillips (1977) exhibits a methodology to compensate for this by utilizing problem-domain or domain-specific knowledge in the inference process. There are certain other special inference paradigms for particular trace classes.

Program inference from protocols Usually, traces mix information about operations applied to data objects, results of tests as to whether predicates hold at certain points during program execution, state snapshots of data values, and other information. Different classes of traces arise if restrictions are placed on the kind of information that may appear in them. Protocols are one such class, in which only operation applications and data structure changes may appear and in which there is no information about control decisions that have been taken during the particular program execution reflected in the trace. An example of a typical protocol for a function that reverses a list would be:

```
input X
X = (A B C)
Y = (A)
X = (B C)
Y = (B A)
X = (C)
Y = (C B A)
output Y
```

Notice that the only information present in the protocol is operation applications and variable state changes. All control information is omitted.

The inference of a program from a collection of protocols involves two phases: (a) constructing a program description that captures the nature of a program and which could have generated a subset of the input protocols, and modifying the program description; and (b) modifying the program description as more protocols become available in order to validate them.

A natural algorithm would then be to hypothesize, by some feature classification process or with the aid of a domain knowledge base, an initial description and then debug it by forcing a unification of the protocol family with the description. The construction of the initial program description can be described as follows:

(1) Match the protocols, that is, find common segments as well as differences by matching their structure.

(2) Find substitutions that unify these protocols. Protocols may differ in variables that have different names, in the same data objects (at the same place in the protocols) having different values, and in differences in the operations that occur. The matching phase produces a set of such differences. The substitution phase finds substitutions that remove these differences. For example, if two protocols refer with different variable names to the same data object, this phase would propose a common name for the two variables. Such substitutions usually take the form constant -> variable or variable-name -> variable-name.

(3) Inductively form loops by detecting repeated equivalent subprotocols. Loop formation is the basic inductive step of this approach.

For example,

```
protocol string = A B C D A B C D
hypothesized loop:
   while <condition>
     do begin
         A;
         B;
         C;
         D;
       end;
```

Since there are infinitely many loop hypotheses for a given protocol, one of the tasks of the system designer is to provide a good set of heuristics to guide the search process during loop formation. For example, one such possible heuristic could be to consider first the loops with minimal nesting level.

(4) Generalize remaining constants to variables.

At this stage, then, a description has been generated where all data object snapshots have an associated variable name, and where loop structures in the program have been inferred. The result of this matching, unification, and abstraction (generalization) process is a semantic net representation of the program.

The next stage is to verify that the hypothesized program description agrees with any additional protocols, and if this is not the case, to modify it. This correction (debugging) phase can be described as follows:

(1) Try to validate new protocols against the program representation--i.e., to symbolically execute the program description to see if it can account for the given protocol.

(2) Find any differences between predicted and actual protocol. The symbolic evaluation process generates a set of differences that are due to the protocol's not matching the program description. This set of differences suggests the kinds of modifications that must be done to the description.

(3) Form a theory for the difference. That is, hypothesize a suitable change to the program description, which removes the particular difference. One way of accomplishing this result is to use a classification process similar to the basic algorithm for inference from examples.

(4) Modify program representation accordingly.

This synthesis paradigm works only for complete protocols, that is, protocols where all data structure changes appear explicitly. Phillips (1977) has proposed a procedure for handling incomplete protocols in a unified framework for synthesis from examples and synthesis from traces or protocols. This procedure is basically as follows: For those segments of a protocol where operations are missing, that is where two states of a data structure appear without intervening operations, the examples component of the system infers a piece of program description (i.e., a sequence of operations) that can take the data object from one state to the other. This program description is nothing but the sequence of missing operation applications. Merging all such sequences with the original incomplete protocol, transforms it into a complete protocol, and the above algorithm for dealing with complete protocols can be used.

Problem-solver generated traces If the domain is fully axiomatized, as may be the case for simple domains like those for robots, it may be possible to synthesize programs from example pairs by using a problem solver that produces a solution to the input pair in the form of a trace.

(1) Synthesize trace from example pair via problem solver.

(2) Using the trace, a set of program schemas for the domain, and a set of schema selection and instantiation heuristics that operate on trace steps, produce a program in terms of domain operators and domain predicates that explain the example pair.

All these paradigms work only for complete traces and protocols. The problem of program inference from incomplete specifications is still under investigation. It is possible that the techniques outlined may be extended to cover the incomplete case by coupling the program synthesizer to a domain-based theory formation module that could, so to speak, "fill in" the missing elements from the original specification. At this point, then, the methodology discussed above could be used.

Traces have the limitations inherent to informal program specifications, namely, the difficulty of specifying the required program uniquely with respect to the limited amount of information conveyed to the synthesizer. Thus, the problem of choosing a good description is left, as a burden, to the user. This problem might be alleviated by the use of greater domain expertise--to produce the program that more nearly resembles the user's desired result.

Traces, and informal specification methods, will be useful for algorithm description and correction in future automatic programming systems. Clearly, the reason for this is that these methods closely reflect the form in which we humans understand and describe programs. Current applications include the synthesis of calculator-like programs from memory-register traces (Biermann & Krishnaswamy, 1974).

References

For theorem proving, see Green, 1969, Waldinger & Levitt, 1974, Kowalski, 1974, Clark & Sickel, 1977; for program transformation, (Darlington & Burstall, 1973), and (Wegbreit, 1976a); for knowledge engineering, (Martin, 1974), (Lenat, 1975), (Biggerstaff, 1976), (Sussman, 1975), and (Green, 1977); for automatic data selection, Low (1978); for traditional problem solving, (Simon, 1972), (Sussman, 1975); for induction from input/output pairs Amarel (1972), Green (1975a), Hardy (1975), Shaw, Swartout, & Green (1975), Siklossy & Sykes (1975), and Summers (1977); for induction from traces, Bauer (1975), Biermann (1972a), Biermann (1976a), Petry & Biermann (1976), Phillips (1977), and Siklossy & Sykes (1975); and for induction from examples, Biermann & Feldman (1970), and Feldman, Gips, Horning, & Reder (1969).

C. PSI

The PSI system is being developed by Cordell Green and his colleagues at Systems Control, Inc., and at Stanford; people who contributed ideas and actually worked on the project include David Barstow, Avra Cohn, Richard P. Gabriel, Jerold Ginsparg, Elaine Kant, Beverly I. Kedzierski, Juan Ludlow, Bruce Nelson, Tom Pressburger, Jorge V. Phillips, Louis Steinberg, Steve T. Tappel, Ronny Van Den Heuval, and Stephen J. Westfold. The goal of the system is the integration of the more specialized methods of automatic programming into a total system. This system then would incorporate specification by examples, by traces, or by interactive natural language dialogue; knowledge engineering; model acquisition; program synthesis; and efficiency analysis. Research objectives include the organization of such a system, the determination of the amount and type of knowledge such a system would require, and the representation of this knowledge.

The program is specified by means of an interactive, mixed-initiative dialogue, which may include as a subpart the specification by example of a trace. Plans are also underway to add specification by means of a loose, very high-level language. The different specification methods can usually be intermixed.

When the specification is interactive natural language dialogue, the user furnishes both a description of what the desired program is to do and an indication of the overall control structure of the program.

The problem area of PSI is symbolic computation, including list processing, searching and sorting, data storage and retrieval, and concept formation.

The overall operation of the system, illustrated in Figure 1, may be divided into two phases: acquisition of a description of the program, and synthesis of the program. During the acquisition phase, several modules of the system--including the parser/interpreter, example/trace, explainer, and moderator--will jointly interact with the user to obtain and construct a net, called the program net, that describes the desired program. Then the program model-builder module converts the net into a complete and consistent description of the program. Afterwards, during the synthesis phase, the coding and efficiency modules, interacting with each other, convert the program model, through the use of repeated transformations, into an efficient program written in the target language.

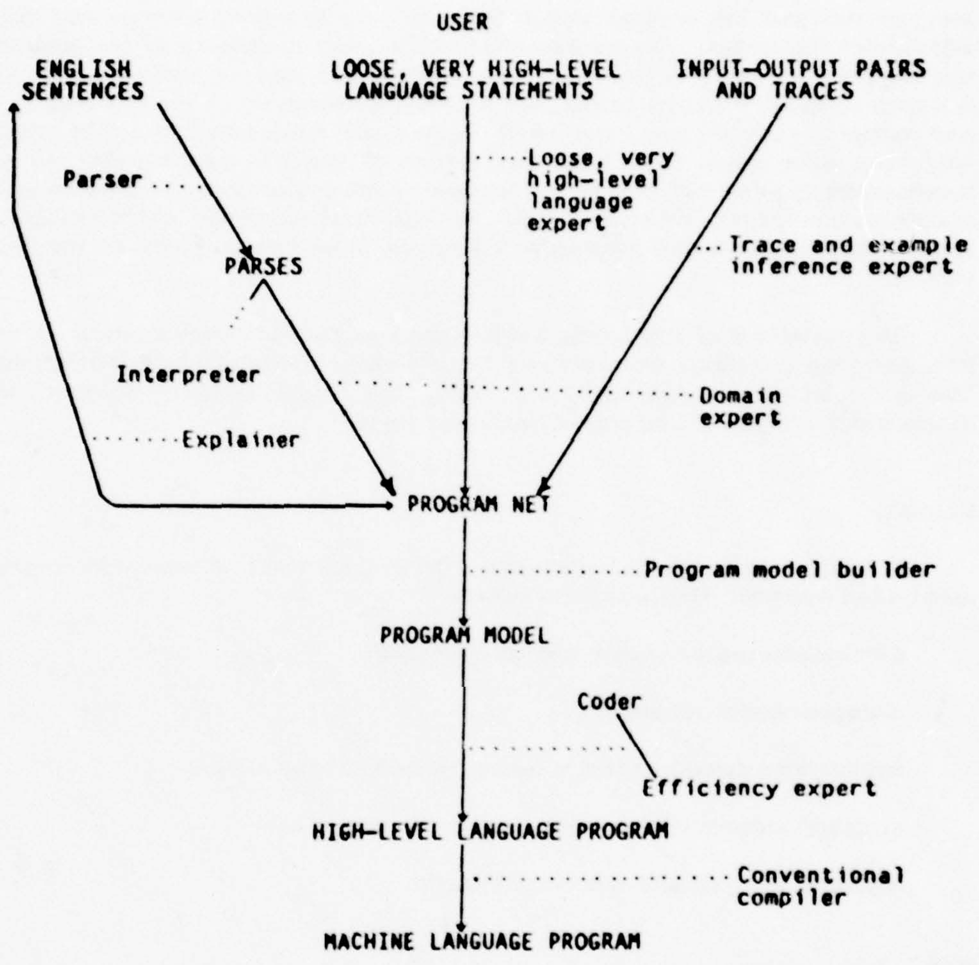


Figure 1: Major paths of information flow in PSI

There were three reasons for separating the operation into acquisition and synthesis phases. First, the problems of designing such a system are more tractable because of the separation. Second, it was envisioned that code generators for different target languages and domain experts for different problem areas could be implemented to result in a versatile modular system. Third, acquisition requires interaction with the user, whereas, in PSI, synthesis does not.

In the overall operation, two of the primary interfaces within the PSI system are the program net and the program model. Both are very high-level program and data structure description languages. The program net forms a looser description of the program than does the program model. Fragments of the program net can be accessed in the order of occurrence in the dialogue, rather than in execution order, which allows a less detailed, local, and partial specification of the program. Since these fragments correspond rather closely to what the user says, they ease the burden of the parser/interpreter as well as the example/trace inference module. As opposed to the program net, the program model includes complete, consistent, and interpretable very high-level algorithmic and information structures. Further description of the program model occurs in the section below on the program model builder.

The remainder of this article briefly describes the PSI modules, presents the status of PSI, and then describes several examples (Figures 2 through 5) from the acquisition phase. The latter includes a specification by interactive natural language dialogue, the resulting program net and model, and a specification by trace.

Experts

PSI is a knowledge-based system organized as a set of closely interacting modules, also called experts. These experts include:

parser/interpreter expert, explainer expert,

dialogue-moderator expert,

applications domain expert, example/trace inference expert,

program model-building expert, coding expert, and the

algorithm analysis and efficiency experts.

Parser/Interpreter

In the acquisition phase, the parser/interpreter expert (Ginsparg, 1978) first parses sentences and then interprets these parses into less linguistic and more program-oriented terms, which are then stored in the program net. This expert efficiently handles a very large English grammar and has knowledge about data structures (e.g., sets, records), control structures (e.g., loops, conditionals, procedures), and more complicated algorithm ideas (e.g., interchanges between the user and the desired program, set construction, quantification). The parser/interpreter can sometimes assign a concept to an unknown word on the basis of the context in which the word appears.

Dialogue Moderator Expert

This expert (Steinberg, 1976) models the user, the dialogue, and the state of the system and selects appropriate questions and statements to present to the user. It also determines whether the user or the expert has the initiative, and at what level on what subject, and attempts to keep PSI and the user in agreement on the current topic. It provides review and preview when the topic changes. This expert decides which of the many questions being asked by the other experts should be passed on to the user. Since experts phrase questions in an internal form based on relations, the dialogue-moderator expert gives questions to the explainer expert which, in turn, converts them into English and gives them to the user.

Explainer Expert

The explainer expert, developed by Richard Gabriel, phrases questions in terms that the user finds meaningful (i.e., in terms related to the problem domain and the previous sentences in the dialogue), rather than using the more programming-oriented terms used in the program net or by the model builder. For example, rather than asking for the definition of "A0018," PSI asks what does it mean for "a scene to fit a concept." The explainer also generates English descriptions of the net.

Example/Trace Expert

PSI also allows specification by traces and examples, since these are useful for inferring data structures and simple spatial transformations. This expert Phillips (1977) handles simple loop and data structure inference and uses several of the techniques discussed in in the last three articles. The final section of this article illustrates how the PSI user can specify part of a program using traces.

Domain Expert

The domain expert, developed by Jorge Phillips, uses knowledge of the application area to help the parser/interpreter and example/trace experts fill in missing information in the program net.

Model Builder

The program model-building expert McCune (1977) applies knowledge of what constitutes a correct program to the conversion of the program net into a complete and consistent program model, which then will be transformed during the synthesis phase into the target language implementation. The model-building expert completes the model by filling in the various pieces of required information and by analyzing the model for consistency; it checks to see that its parts are legal both with respect to each other and with respect to the semantics of the program-modeling language. Information is filled in either by default, by inference mechanisms (which are in the form of rules and which make use of consistency requirements), or by queries to other experts, which may eventually result in a query to the user. As an example, suppose that the program net contains "x part of y" and that the model builder needs to fill in whether "part of" is to mean set membership, subset inclusion, component of y, the image of x under some correspondence relation with y, or whether there might be an unspecified intervening subpart. Such information may be deducible from the structures of x and y, if these structures are known or when they become known.

The model builder also corrects minor inconsistencies, adds cross-references, and generalizes parts of the program description so that the synthesis phase has more freedom in looking for a good implementation. Thus, if the program net specifies that a certain object is to be a set of ordered pairs, the program model may, if appropriate, indicate that the object is to be a correspondence (i.e., a functional mapping).

Coding and Efficiency Experts

These two experts are responsible for the synthesis phase. The coding expert's knowledge base contains rules that transform parts of a program description to forms closer to the target language. It is the goal of the efficiency expert to guide the choice of the different rules so that an efficient target language implementation eventually results. These two experts, also known as the systems LIBRA and PECOS, are discussed more explicitly in articles devoted to their description. See the articles D8D5.

Status

The entire PSI system can now be used by a knowledgeable user. PSI has successfully produced LISP code that implements the specifications given in English dialogues for four variants of CLASS, a simple pattern classification program. In addition, other tasks are handled by individual experts in a stand-alone mode; see, for example, D5 D9.

The next section concludes this article by describing examples of PSI at work.

Examples

Figure 2 illustrates an interactive natural language dialogue held with the PSI system in which the user specifies a simple program, to be called CLASS. CLASS includes many programming constructs necessary for more complex programs.

```
What is the name of the program you want to write?
** CLASS.

Does CLASS have a special program type?
** Classification.

Please describe CLASS.
** CLASS first inputs a concept. Then it repeatedly accepts an input
**scene from the user, classifies it based on whether or not it fits the
**concept, and outputs this classification to the user.

What is the structure of the concept?
** A concept is a collection of relations. Each relation is just a string.

What is the exit test of the loop?
** Stop when the user types "QUIT" instead of a scene.

What is the structure of the scene?
** The same as the concept.

Please define what it means for a scene to fit a concept.
** A scene fits a concept if all the relations in the concept are
** in the scene.

What is the format of the classification output?
** Output "FIT" if the scene fit, "DIDN'T FIT" if it didn't.

Are we finished?
** Yes.
```

Figure 2. Specification by interactive natural language dialogue: CLASS

The user intends that the CLASS program input a sample set of items (e.g., a list of job qualifications) and then repeatedly input a trial set (e.g., the qualifications of an applicant), each time testing whether the sample set (required qualifications) is a subset of the trial set (applicant qualifications) and printing "FIT" or "DIDN'T FIT," accordingly. The user further intends that a person will be able to terminate CLASS simply by typing the word "QUIT," instead of a trial set.

Based upon its understanding of the dialogue, the parser/interpreter expert produces the program net, which is summarized in Figure 3 (the algorithmic part of the net is shown in an ALGOL-like notation). Then the program model-building expert creates the very high-level complete and consistent model of Figure 4. After repeated application of transformation rules during the synthesis phase, the coding and efficiency experts will convert this model into an efficient target language implementation.

```
A2 is either a set whose generic element is a string or a string whose
     value is "QUIT".
A1 is a set whose generic element is a string.
A4 is the generic element of A1.
A3 is either TRUE or FALSE.

B1 is a variable bound to A2.
B2 is a variable bound to A1.
B3 is a variable bound to A4.


CLASS
  PRINT("Ready for the CONCEPT")
  A1 ← READ()
LOOP1:
  PRINT("Ready for the SCENE")
  A2 ← READ()
  IF EQUAL(A2,"QUIT") THEN GO_TO EXIT1
  A3 ← FIT(A2,A1)
  CASES:  IF A3 THEN PRINT("FIT")
    ELSE IF NOT(A3) THEN PRINT("DIDN'T FIT")
  GO_TO LOOP1

 EXIT1:

 FIT(B1,B2)
   FOR_ALL B3 IMPLIES(MEMBER(B3,B2),MEMBER(B3,B1))
```

Figure 3. Summary of the program net.

```
program CLASS;
  type
    a0032 : set of string ,
    a0053 : alternative of [<string = >"QUIT" , a0032];
  vars
    a0011 , a0014 , a0035 , a0036 : a0032 ,
    a0055 , m0080 : a0053 ,
    m0095 : string = "DIDN'T FIT" ,
    m0092 : string = "FIT" ,
    m0091 : Boolean ,
    m0081 : string = "QUIT" ;
  procedure a0067(a0036 , a0035 : a0032) : Boolean ;
    a0035 K a0036 ;
  procedure a0065(a0055 : a0053) : Boolean ;
    a0055 = "QUIT" ;
    begin
      a0011 ← input(a0032 , user , "READY FOR CONCEPT" ,
       "Illegal input.  Input again:  ") ;
      until A0051
      repeat
        begin
        m0080 ← input(a0053 , user , "READY" , "Illegal input.
                                            Input again: ");
        if a0065(m0080) then assert_exit_condition(A0051) ;
          a0014 ← m0080 ;
          m0091 ← a0067(a0014 , a0011) ;
          case
            & m0091 : inform_user("DIDN'T FIT") ;
             m0091 : inform_user("FIT") ;
          endcase
        end
      finally
      A0051 :
      endloop
  end ;
```

Figure 4. The program model.

Traces are another method of specification allowed by the PSI system. Figure 5 shows the use of a trace to specify part of the behavior of a program called TF ("Theory Formation"). A simplified version of Pat Winston's concept formation program,(Winston, 1975), TF builds and updates an internal model of a concept. A concept is a collection of "may" and "must" conditions. TF builds and updates the model by repeatedly reading in a scene, guessing whether the scene is an instance of the concept, verifying with the person using TF whether the guess was correct or incorrect, and updating the model of the concept accordingly. The trace in Figure 5 shows the specification for only a part of the behavior of TF, the part that describes how TF is to update the model, given that a scene does or does not fit a concept. The other parts of TF can be specified by trace or by interactive natural language dialogue.

Concept:	[]
Scene:	[(block a)(block b)(on a b)]
Result of fit:	True
Updated concept:	[((block a) may)((block b) may)((on a b) may)]
Concept:	[((block a) may)((block b) may)((on a b) may)]
Scene:	[(block a)(block b)]
Result of fit:	False
Updated concept:	[((block a) may)((block b) may)((on a b) must)]
Concept:	[((block a) may)((block b) may)((on a b) must)]
Scene:	[(block a)(block b)(block c)(on a b)]
Result of fit:	True
Updated concept:	[((block a) may)((block b) may)((block c) may) ((on a b) must)]

Figure 5. A specification by trace.

From this specification, the example/trace inference expert generates the following information about the desired program: If the scene fits the concept, then add all relations in the scene but not present in the concept to the concept and mark them with "may." Otherwise, if the scene doesn't fit the concept, then change the marking of all relations marked "may" in the concept and not appearing in the scene from "may" to "must."

References

See Barstow (1977a), Barstow (1977b), Barstow (1977c), Barstow & Kant (1977), Ginsparg (1978), Green (1975a), Green (1975b), Green (1976a), Green (1976b), Green (1976c), Green (1977), Green (1978), Green & Barstow (1975), Green & Barstow (1977a), Green & Barstow (1977b), Green & Barstow (1978), Kant (1977), Kant (1978), McCune (1977), Phillips (1977), and Shaw, Swartout, & Green (1975).

D. SAFE

The SAFE system, developed at USC Information Sciences Institute by Robert Balzer, Neil Goldman, David Wile, and Chuck Williams (with the recent addition of Lee Erman and Phil London), accepts a program specification consisting of pre-parsed English, with limited syntax and vocabulary, including terms from the problem domain. The phrases and sentences of this specification, however, may be ambiguous and may fail to explicitly provide all the information required in a formal program specification. Therefore, using a large number of built-in constraints (that must be satisfied by any well-formed program), any specified constraints on the problem domain, and an occasional interaction with the user, SAFE resolves ambiguities, fills in missing pieces of information, and produces a high-level, complete program specification. To decide on missing pieces of information, SAFE uses a variety of techniques, including backtracking (see article AI Languages) and a form of symbolic execution.

The SAFE system views the task of Automatic Programming as the production of a program from a description of the desired *behavior* of that program. There are four major differences between a conventionally specified program and a program described in terms of its desired behavior.

Informality: The behavioral description is informal. It contains ambiguity (alternative interpretations yielding distinct behaviors) and "partial" constructs (constructs missing pieces of information that must be supplied before any interpretation is possible). A conventionally specified program, on the other hand, is formal; its meaning is completely and unambiguously defined by the semantics of the programming language.

Vocabulary: The primitive terms used in the behavioral description are those of the problem domain. General-purpose programming languages, on the other hand, provide a primitive vocabulary that is significantly more independent of particular problem areas.

Executability: Informality aside, it is possible, and sometimes desirable, to describe behavior in terms of relationships between desired and achieved states of a process , rather than by rules that specify how to obtain the desired state. Conventionally specified programs must specify an algorithm for reaching the desired state.

Efficiency: Conventionally specified programs contain many details of operation beyond the desired input/output behavior. Among these are data representation, internal communication protocols, store-recompute decisions, etc., that affect a program's efficiency (utilization of computer resources and time). In general, these details should not appear in the description of input/output behavior.

When one writes a program in the conventional manner, one must formalize the behavioral specification, translate the terms of the problem domain into those of a general programming language, guarantee that the specified algorithms actually achieve the desired results, and make a myriad of decisions for the sake of an efficient implementation.

The ISI group has attempted to split the task of creating a program into two separate parts by designing a formal, complete specification language (Balzer & Goldman, 1979) that allows behavioral specifications to be stated in terms specific to the problem domain while avoiding efficiency and representational concerns. This formal specification language acts as an interface between two projects that deal respectively with the first issue, translation from informal to formal specifications, and the last issue, optimization of a formal specification. The former project is the subject of this article, while the latter is described elsewhere (Balzer, Goldman, & Wile, 1976). The other issues, domain-specific vocabulary and executability, are addressed within the formal specification language.

The SAFE project has concentrated on only the first of the above specification issues: automatically producing a formal description from an informal description. It is not, therefore, a complete *automatic programming system*. The user of the SAFE system provides a behavioral description in a pre-parsed, limited subset of English, including terms from the problem area. SAFE then seeks to determine a way of resolving all ambiguities and of filling in all missing information in a way that satisfies SAFE's knowledge of the constraints that all programs must satisfy. The result is a complete, unambiguous, very high-level program specification in a language called AP2.

Partial Descriptions

After studying many examples of program specifications written in English, the SAFE research group concluded that the main semantic difference between these specifications and their formal equivalent is that partial descriptions rather than complete descriptions were used. When such partial descriptions were used, it was because the missing information could be determined from the surrounding context. These partial descriptions possess some of the useful properties of natural language specifications that are lacking in formal languages. They focus both the writer's and reader's attention on the relevant issues and condense the specification. Furthermore, the extensive use of context almost totally eliminates bookkeeping operations from the natural language specification.

A partial description may have zero or one or more valid interpretations in a given context. If a single valid interpretation is found for a description, it is unambiguous in that context. Multiple valid interpretations indicate that there is not sufficient information from the context to complete the description and that interaction with the user is required to resolve the ambiguity. If a partial description possesses no valid interpretation, it is inconsistent within the existing context.

The SAFE system incorporates the most prevalent forms of partial descriptions found in natural language specifications:

Partial sequencing: Operations are not always described in the order of execution. While sequencing may sometimes be described explicitly, it is frequently implicit in the relationships between operations. Example: "Output generated while compiling is sent to a scratch file. This file must be opened in *write only* mode. (file should be opened before compiling commences)."

Missing operands: The operands of operations are frequently omitted because they are recoverable from context. Recovering them may involve considering the operation's definition, other operands, and the procedural context. Example: "Do not mount a tape for a job unless the tape drive has been assigned (to that job)."

Incomplete reference: A description of an object(s) may match several objects whereas it was intended to refer to only one or possibly a subset of these objects. A complete description may be recovered by methods similar to that for missing operands. Example: "When the mail program starts, it opens the file named MESSAGE (in the directory of the job running the program)."

Type coercions: Often, people using natural language do not precisely specify the object intended, but instead specify an associated object or a subpart of an object. This situation can be recognized by a mismatch between the type of object actually specified and the type of object expected. Example: "Information messages are copied to each logged-in user (to the terminal of the job of each logged-in user)."

Operation of SAFE

The goal of SAFE is to complete the various partial descriptions in the user's specification of the desired program so as to produce a formal specification of the whole program. SAFE goes through several phases, but in all phases the system uses a variety of constraints to achieve the goal of completing the partial descriptions. These include built-in criteria that any formal program must meet (e.g., information must be produced before it is consumed), built-in heuristics that "sensible" programs will meet (e.g., the value of a conditional must depend on the program data), as well as any known or discovered constraints particular to a program's domain (e.g., each file in a directory has a distinct name). In fact, since programs are highly constrained objects, there are a large number of constraints that any "well-formed" program must satisfy, and this is one reason programs are hard to write.

In general, each partial description has several different possible completions. Based on the partial description and the context in which it occurs, an ordered set of possible completions is created for it. But one decision cannot be made in isolation from the others; decisions must be consistent with one another and the resulting program must make sense as a whole, satisfying all the criteria of well-formed programs.

The problem of finding viable completions for a collection of partial descriptions provides a classical backtracking situation, since there are many interrelated individual decisions that, in combination, can be either accepted or rejected on the basis of the constraints. SAFE utilizes the constraints so that early rejection possibilities can be realized.

The operation of SAFE consists of three sequential phases: the linguistics, planning, and meta-evaluation phases. The cumulative effect of these phases is to produce a formal specification that is composed of declarative and procedural portions. The declarative part, or domain model, specifies the types of objects manipulated by the process, the various ways they may relate to one another, the actions that may be performed on various object types, and other global regularities of the problem domain. The procedural portion specifies the controlled application of actions to objects.

The linguistic phase, using production rules, transforms the parse trees of the English specification into fragments that retain the semantic content while discarding the syntactic detail. The production rules capture many context-sensitive aspects of natural language such as various uses of the verb "be" and of quantifiers. The production rules may also add declarations to the domain model, with user approval, when this is required for interpretation of the input. This procedure is accomplished by distinguishing two sets of conditions on each rule: those relating to the linguistic form of the phrase being processed, and those relating a form to the domain model. If the linguistic form conditions are not satisfied (e.g., a clause using a transitive verb) but the domain model conditions are (e.g., the verb names an action in the problem domain that has operands of types compatible with the verb arguments), then the domain model conditions are assumed.

The planning phase determines the overall sequencing of the operations in the program. It also determines which fragments belong together and how they are to interact. It does this by using explicit sequencing information in the description, such as "A is executed immediately after B," "A is invoked whenever the condition C becomes true," as well as static flow constraints on well-formed processes such as:

Before information is consumed (used by one fragment), it must be produced (created by the same or another fragment).

Expected outputs of the whole program or of a subprogram must be produced somewhere within that program.

The results of each described operation must be used or referenced somewhere.

The final phase, meta-evaluation, uses dynamic constraints to help determine the proper completion of partial descriptions. Dynamic constraints are those that apply, or at least relate to, the program during execution. Examples of such constraints are:

It must be possible (in general) to execute both branches of a conditional statement (otherwise why would the user have specified a conditional).

The constraints of a domain must not be violated.

Since no actual input data is available for testing the execution of the program and since the program must be well-formed for all allowable inputs, inputs are represented symbolically. Instead of actual execution, the program is symbolically executed on the inputs, which provides a much stronger test of the constraints than would execution on any particular set of inputs. The result is a database of relationships between the symbolic values and, implicitly, a database of relationships between program variables that are bound to these values.

All decisions concerning the proper interpretation of partial descriptions that affect the computation to some point in the execution (but not beyond) must be made before these dynamic criteria can be tested at that point in the execution. Thus, decisions are made as they are needed by the computation of the program, and the symbolic state of the program is examined at each stage of the computation. This arrangement allows the dynamic state-of-computation criteria to be used to obtain early rejection of infeasible alternatives.

There is an additional point worth noting. Representing the complete state of a computation during symbolic execution is very difficult (e.g., it is quite hard to determine the state after execution of a loop or conditional statement) and more detailed than necessary for testing the constraints. Therefore, the SAFE system uses a weaker form of symbolic interpretation called Meta-Evaluation, which only partially determines the program's state as the computation proceeds (e.g., loops are executed only once for some "generic" element).

Notice that symbolic execution requires that the sequential relationships between the fragments be known; therefore the meta-evaluation phase must follow the planning phase.

Finally, the global referencing constraints (such as "The body of a procedure must make use of the procedure's parameters") test the overall use of names within the program and, thus, cannot be tested until all decisions have been made. These criteria can be tested only after the Meta-Evaluation is complete.

Status

The prototype system has successfully handled the 75-200 word specifications of three quite distinct programs. In these cases the SAFE output of a completed specification, including domain structure definition, requires approximately two pages. One example concerned part of a system for scheduling transmissions in a communications network. Given a table (SOL) containing entries for various network subscribers and for various unassigned time slots (RATS), a schedule of absolute times when a particular subscriber could broadcast on the network was tabulated. The input specification to SAFE is:

```
((THE SOL)
 (IS SEARCHED)
 FOR
 (AN ENTRY FOR (THE SUBSCRIBER)))

(IF ((ONE)
    (IS FOUND))
   ((THE SUBSCRIBER'S (RELATIVE TRANSMISSION TIME))
    (IS COMPUTED) ACCORDING TO ("FORMULA-1")))


((THE SUBSCRIBER'S (CLOCK TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING-TO ("FORMULA-2")))

 WHEN ((THE TRANSMISSION TIME))
      (HAS BEEN COMPUTED))
     ((IT)
      (IS INSERTED)
      AS (THE (PRIMARY ENTRY))
      IN (A (TRANSMISSION SCHEDULE))))

FOR (EACH RATS ENTRY)
   (PERFORM)
   (: ((THE RATS'S (RELATIVE TRANSMISSION TIME))
      (IS COMPUTED) ACCORDING TO ("FORMULA-1"))
```

```
        ((THE RATS'S (CLOCK TRANSMISSION TIME))
         (IS COMPUTED) ACCORDING TO ("FORMULA-2"))))

    ((THE RATS (TRANSMISSION TIMES))
     (ARE ENTERED)
     INTO (THE SCHEDULE))
```

Figure 1. Actual input for link scheduling example.

In formalizing this description, SAFE encountered and resolved the following characteristics of *informal specifications*:

number of missing operands	= 7
number of incomplete references	= 12
number of implicit type coercions	= 3
number of implicit sequencing decisions	= 4

Robustness of the system has been increased by processing a number of perturbations of each of the major examples. These have involved specifying the same process but varying the syntax and vocabulary used, the partial descriptions used, and the formal knowledge provided about the problem domain.

Future Developments

The key technical restrictions of the prototype system appear to be (a) the sequential application of the three phases, which prohibits adequate interactions between the expertise embodied in each, and (b) the backtracking within the meta-evaluation phase, which corresponds to restarting the symbolic execution from an earlier point, which can lead to much unnecessary search. To correct these limitations, a reformulation of the system architecture within a framework derived from the HEARSAY II speech understanding system (see article Speech.C) is currently in progress. This framework consists of a number of cooperating experts interacting via a "blackboard" database.

Simultaneously, the system is being scaled up to handle larger practical specifications (approximately 20 pages). Later, the project will consider the formalization of incremental informal specifications so that it can also provide help during both specification formulation and maintenance activities.

References

See Balzer, Goldman, & Wile (1976), Balzer, Goldman, & Wile (1977a), Balzer, Goldman, & Wile (1978), and Balzer & Goldman (1979).

E. Programmer's Apprentice

The Programmer's Apprentice (PA) is an interactive system for assisting programmers with the task of programming. It is being designed and implemented at MIT by Charles Rich, Howard Shrobe, and Richard Waters. Currently, most, but not all, of the modules that comprise the PA system are running. It should be kept in mind that the scenario described here illustrates the projected operation of the system, not the present operation. The intent of the PA is that the programmer will do the hard parts of design and implementation, while the PA will act as a junior partner and critic, keeping track of details and assisting the programmer in the documentation, verification, debugging, and modification of his program. In order to cooperate with the programmer in this fashion, the PA must be able to "understand" what is going on. From the point of view of Artificial Intelligence, the central development of the Programmer's Apprentice project has been the design of a representation (called a "plan") for programs and for knowledge about programming that serves as the basis for this "understanding." Developing and reasoning about plans is the central activity of the PA.

The "plan" for a program represents the program as a network of operations interconnected by links explicitly representing data flow and control flow. The advantage of this aspect of the plan formalism is that it abstracts away from the specific syntactic constructs used by various programming languages in order to implement control flow and data flow. The most novel aspect of the plan formalism is that it goes beyond this level in order to create a vehicle for expressing the *logical* interrelationships in a program. First, a plan is not just a graph of primitive operations. Rather, it is a hierarchy of segments within segments, where each segment corresponds to a unit of behavior and has an input/output specification that describes features of this behavior. The plan specifies how each nonterminal segment is constructed out of the segments contained within it. This segmentation is important because it breaks the plan up into localities that can be understood in isolation from each other. Second, the behavior of a segment is related to the behavior of its subsegments. This interrelationship is represented by explicit dependency links that record the goal-subgoal and prerequisite relationships between the input-output specification for a segment and those for its subsegments. Taken together, the links summarize a proof of how these specifications for a segment follow from the specifications of its subsegments and from the way the subsegments are interconnected by control flow and data flow. A final aspect of the plan formalism is that there may be more than one plan for a given segment of a program, with each plan representing a different point of view on the segment. The data structures used by a program are represented by specifying their parts, properties, and the relationships between them in a method similar to data abstractions (Zilles, 1975; Liskov, 1977).

Knowledge about programming in general is also represented using plans and data structure descriptions. This knowledge is stored in the PA in a database of common algorithms and data structure implementations called the "plan library." The PA's "understanding" of a program is embodied in a hierarchical plan for it. In general, the subplan for each individual segment in terms of its subsegments will be an instance of some plan stored in the plan library. This structure gives the PA access to all of the information stored in the plan library about the particular subplan as soon as it can make a guess as to what the subplan is.

A Scenario of Use of the Programmer's Apprentice

The following imagined conversation between a programmer and the PA is presented in order to illustrate the intended operation of the system. (Comments discussing the scenario are printed in *italics*.) The scenario illustrates the following four basic areas in which the PA can assist a programmer:

(1) **Documentation**: One of the primary services the PA provides is automatic, permanent, and in-depth documentation of the program. The PA remembers not only explicit commentary supplied by the programmer with the code, but also a substantial body of derived information describing the logical structure underlying the program, such as the dependency relationships between parts of the program.

(2) **Verification**: The development of a program is accompanied by the construction of a sequence of plans at various levels of abstraction. At each step, the PA attempts to verify that the current plan is both consistent and sufficient to accomplish the desired goal. As more information is specified, the PA's reasoning about these plans approaches a complete verification of the program.

(3) **Debugging**: Any discrepancy between the PA's understanding of the programmer's intent and the actual operation of the program is reported to the programmer as a potential bug.

(4) **Managing Modification**: Perhaps the most useful aspect of the PA is that it can help a programmer modify his program without introducing new bugs. Based on its knowledge of the logical relationships between parts of a program, the PA is able to determine what parts of a program can be affected by a proposed change, and how they can be affected. It can use this information to warn the programmer of impending difficulties.

The scenario traces the design, coding, and subsequent modification of a program that deletes an entry from a hash table. The scenario picks up in the middle of a session, at a point where the programmer has already made many design choices and conveyed them to the PA. In particular, he has stated the input-output specifications for the subroutine DELETE and chosen the basic data structures to be used. The hash table is implemented as an array of linked lists called "buckets." Each of the entries in a bucket is a dotted-pair whose CAR is the key for the entry.

Programmer: The plan for the routine DELETE has three main steps: The first one hashes the input key to get an index; the second one fetches the bucket corresponding to that index from the table; and the third splices out the entry in the bucket whose key part is equal to the input key.

The only feature of this scenario that is not a current research goal of the Programmer's Apprentice project is the use of natural English dialogue. Although the system should be able to support a sophisticated natural language "front-end," no effort has been expended in this direction to date. As a result, the programmer would actually type in something like:

```
"(subsegments-of-DELETE step1-step2-step3)
(specifications-for-step1 (inputs-key1) (outputs-index)
      (assert-(=-index-(hash-key1)))) ..."
```

The PA uses the programmer's outline of the design in order to build a plan representing the program's intended structure. References to basic programming techniques such as "splicing out" (deleting elements from a linked list) trigger the retrieval of plans from the plan library.

PA: I can't verify that this plan satisfies the specifications for DELETE. In particular, when the input key is equal to the key part of the first entry in a bucket, I can't prove that the entry is no longer in the table after DELETE is executed. Note that in this situation, the bucket after splicing out entries has a different initial cell from the bucket before splicing out entries, and that therefore the table isn't changed.

After constructing a plan, the PA uses its reasoning abilities in order to check for bugs. Above, the PA finds one and reports it to the programmer. In general, it is the responsibility of the programmer to supply corrections. Below, the programmer corrects the plan by issuing semantically based editing commands to the PA.

Programmer: Modify the plan as follows. Add a final step that compares the bucket after splicing with the bucket before splicing and stores the new bucket in the table under the index if the two initial cells are not identical.

PA: OK, the plan now seems correct.

The programmer then types in the code for DELETE. In cases where the programmer specifies the plan for a program as completely as he does here, he could ask the PA to write the code. It is expected that the programmer will actually only write code in situations where it is easier to write the code than to fully explain the plan.

```
Programmer: (DEFUN DELETE (KEY) ; hash table delete routine.
              (PROG (BKT1 BKT2)
                  (SETQ BKT1 (TBL (HASH KEY)) BKT2 (BKT-DELETE KEY BKT1))
                  (COND ((NOT (EQ BKT1 BKT2))
                                ; check if first entry in bucket deleted.
                        (STORE (TBL (HASH KEY)) BKT1)))))
```

The PA analyzes the code written by the programmer in order to construct a plan for it. Comparing this plan with the plan directly described by the programmer, the PA discovers an implementation bug: In the last line of code the user probably meant to store into the hash table not the first bucket, BKT1, but the second bucket, BKT2.

PA: This code is not consistent with your plan. When the first entry is deleted, the code stores the bucket before splicing, rather than the

bucket after splicing. Did you mean

"(STORE-(TBL-(HASH-KEY))-BKT2)"?

Programmer: Yes, fix the code. Also, use the standard list deletion algorithm to code BKT-DELETE so that it splices out of the bucket all entries whose key equals the indicated value.

```
PA:  (DEFUN BKT-DELETE (KEY BUCKET) ; list deletion routine coded by the PA.
     (PROG (CURRENT PREVIOUS)
         (SETQ CURRENT BUCKET   PREVIOUS NIL)
       LP (COND ((NULL CURRENT) (RETURN BUCKET))
              ((EQUAL (CAAR CURRENT) KEY)
                             ; if keypart equal, splice out entry.
               (COND (PREVIOUS (RPLACD PREVIOUS (CDR CURRENT)))
                     (T (SETQ BUCKET (CDR BUCKET)))))
                             ;special if first element deleted.
              (T (SETQ PREVIOUS CURRENT)))
         (SETQ CURRENT (CDR CURRENT))
         (GO LP)))
```

The PA is able to code a program like this itself, even though the programmer did not specify a plan for it, because it is an implementation of a standard algorithm that corresponds to a plan in the plan library. Code written by the PA can be freely mixed with code written by the programmer and each can edit the other's work.

Some time later it is necessary to modify the representation of the buckets. The PA helps the (perhaps new) programmer maintain consistency.

Programmer: I want to make a change in the representation of buckets in the hashing programs. A bucket now has two parts: a count field and a list of entries as before. The count field will contain an integer equal to the length of the list of entries. Let a bucket be a dotted-pair whose CAR is the count and whose CDR is the list of entries.

When the PA inspects the logical dependencies in the plans, it discovers that a type requirement has been violated.

PA: The proposed change will affect DELETE as follows: After the bucket is fetched, it is passed to BKT-DELETE which does the actual deletion. However, BKT-DELETE requires a list of entries while in this new implementation, a bucket is no longer just a list of entries.

Programmer: Explain the plan for BKT-DELETE please.

PA: BKT-DELETE is implemented following the standard list deletion plan. The inputs are a key and a list of entries. The plan is a search loop using two pointers: a pointer to the current entry, which is initialized to the input list, and a trailing pointer, which is initialized to NIL. On each iteration the loop tests the key of the current entry. If it is equal to the input key, the element is spliced out of the list by RPLACDing the trailing pointer.

Acting in its role as documentation, the PA describes the structure of BKT-DELETE by reporting out the structure of its plan. The programmer then decides how to modify BKT-DELETE.

Programmer: Change the specifications of BKT-DELETE so that it expects as input a bucket of the new type, rather than a list of entries. Also change the initialization of the pointers to read: "(SETQ-PREVIOUS-BUCKET- CURRENT-(CDR-BUCKET))".

The PA reanalyzes the program in order to verify that this change fixes the problem without causing any other problems to arise. This analysis reveals the presence of inaccessible code which the PA reports to the programmer.

PA: OK this fixes the problem. However, now the special case tests in BKT-DELETE and DELETE are not necessary since PREVIOUS is never NIL and therefore elements are always deleted by side-effect.

... and so the session continues with the PA looking over the programmer's shoulder.

Operation of the System

The design of the PA is based on four modules, a surface analyzer, a recognizer, an interactive module, and a deductive module; and two data bases, the plan library and a scratch pad called the "design notebook." Only the first three modules have been implemented so far. As described above, the plan library contains the PA's knowledge of programming in general. The design notebook contains the PA's evolving knowledge of the particular programs being worked on and serves as the communication center for the system as a whole. The modules communicate with one another solely by making assertions in the design notebook. Each module has predefined trigger patterns which cause it to perform specific tasks (such as making a deduction or querying the user) whenever appropriate assertions appear in the notebook. Every assertion added to the notebook is also accompanied by a justification of its presence. These justifications make it possible for the PA to account for its actions.

The surface analyzer is used to construct simple surface plans for sections of code written by the programmer. It is the only module whose implementation depends on the particular programming language being used. To date, surface analyzers have been implemented for both LISP and FORTRAN. The recognition module takes over where the surface analyzer leaves off in order to construct a detailed plan for a piece of code. It first breaks up the surface plan by identifying weakly interacting subsegments that can be further analyzed in isolation from each other. It then compares these subsegments with the plans in the library in order to determine more detailed plans for the program.

The interactive module is the communication link between the PA and the programmer.

It converts the programmer's input (which can consist of code, direct specification of a plan, or various requests) into assertions in the design notebook and decides what to say to the programmer based on the information currently in the notebook. The deductive module operates in the background in cooperation with all of the other modules. It performs the deductions necessary to verify a proposed match between a program and a plan, to detect bugs in a plan, and to determine the ramifications of a proposed modification to a program or plan.

At a given moment, the design notebook holds the sum total of what the PA knows about the program being worked on. This information triggers additional activity by the modules. If the recognizer and deductive modules are strong enough and the program is simple enough, this process will culminate in a complete understanding and verification of the program. However, typically, this will not be the case, and some questions (such as the exact plan for a segment or the correctness of a specification) will remain unresolved in the notebook. The flexible architecture chosen for the PA makes it possible for the PA to exhibit useful partial performance in this situation. It is able to ignore what it doesn't understand and work constructively with what it does understand. The programmer can be called upon to fill in the gaps.

Current Status of the Programmer's Apprentice

Rich and Shrobe (1976) laid out the basic idea of a plan and the initial design of the PA. Since that time Rich, Shrobe, and Waters have been working together on further aspects of the theory along with design and implementation of the PA.

Rich's work (forthcoming) centers on the plan library and the recognition process. He is using the plan representation in order to codify a large body of common programming strategies in the domain of non numerical programming. He is also designing a recognition module that will be able to identify instances of plans in the library as they occur in combination in a programmer's program.

Shrobe (1978) has implemented a prototype deductive module that can reason about programs represented by plans. An important aspect of its operation is that it maintains a record of the dependency relationships embodied in its deductions. In doing this it builds up some of the logical structure that is a vital part of a plan for a program. He is currently designing an improved version of this deductive module.

Waters (1976, 1978) has implemented a system that can analyze the code for a program and produce the basic structure of a plan for the entire program. The system corresponds to the surface analysis module and the initial phase of the recognition process. The basic idea behind Waters' work is that plans for typical programs are built up in a small number of stereotyped ways and that features in the code for a program can be used to determine how the plan for the program should be built up.

The goal for the immediate future is to construct a prototype system that can exhibit the kind of behavior shown in the scenario. To do this, an interactive module must be built, and the other modules must be connected together into an integrated system. Looking further ahead, additional modules (such as a simple program synthesis module, and one dealing with efficiency issues) will be added to the PA, and the existing ones will be strengthened so that the PA can assume an even larger part of the programming process.

References

See Liskov et al. (1977), Rich & Shrobe (1976), Rich & Shrobe (1978), Rich (1979), Shrobe (1978), Waters (1976), Waters (1978), Waters (1979), and Zilles (1975).

F. PECOS

Developed in 1976 by David Barstow (Barstow,1976), the automatic programming system PECOS serves as the coding expert of Standford's PSI project (see article D2 and Barstow, 1979. The foundations of PECOS are based on ideas presented in Green & Barstow (1977a), and Green & Barstow (1978). Though PECOS can act in conjunction with the PSI system, it can also stand on its own and interact directly with the user. The original problem area of PECOS was symbolic programming, which includes simple list processing, sorting, database retrieval, and concept formation. This domain has recently been extended to graph theory and simple number theory. Programs are specified in terms of very high-level constructs including *data structures*, like collections or mappings, and *operations*, like testing for membership in a collection or computing the inverse image of an object under a mapping. Knowledge about programming in the problem area has been codified (i.e., made explicit and put into machine useable form) primarily in the form of transformation rules, and these have been entered into PECOS's knowledge base. Most of the rules describe how constructs and operations can be represented or implemented in terms of other constructs and operations that are closer to, or actually in, the target language LISP (actually a subset of INTERLISP, Teitelman et al., 1978). These rules can identify design decisions and can also serve as limited explanations.

The operation of the system proceeds by the repeated selection and application of the transformation rules in the knowledge base to parts of the program. Also referred to as *gradual refinement*, this transformation process reduces the high-level specification to an implementation fully within the target language. Each application of a rule is said to produce a partial implementation or *refinement* of the program, and the transformation rules are called *refinement* rules.

Conflict Resolution

At some points during the transformation process, a conflict may arise because several rules apply to the same part of the program. The handling of this situation is important: The application of the several rules ultimately results in different target language implementations that often vary significantly in terms of efficiency. There are three ways to handle this situation.

(1) If PECOS is interacting directly with the user, the user may select which rule should be applied (and thus which implementation will be constructed).

(2) For the convenience of the user, PECOS can choose one of the applicable rules, using about a dozen heuristics it has to pick the rule that leads to the *more* efficient implementation. These heuristics handle about two-thirds of the choices that typically arise.

(3) When no heuristic applies and the user is uncertain about which rule is "best" for his or her purposes, PECOS can apply each in parallel, constructing a separate implementation for each rule applied.

When PECOS functions as the Coding Expert of the PSI program synthesis system (Green, 1976b;D2), choices between rules are made by an automated Efficiency Expert known as LIBRA (see article D9, Kant (1977)), which incorporates more sophisticated analytic techniques than the simple heuristics used by PECOS. The capability of developing different implementations in parallel is used extensively in the interactions between PECOS and LIBRA (Barstow & Kant, 1977).

PECOS's Knowledge Base

PECOS's knowledge base consists of about 400 rules dealing with a variety of symbolic programming concepts. The most abstract concepts are those of the specification language (e.g., collection, inverse image, enumerating the objects in a collection, etc.). The implementation techniques covered by the rules include the representation of collections as linked lists, arrays (both ordered and unordered), and Boolean mappings, and the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings (indexed by range element). As a natural by-product, these rules also cover sorting within a transfer paradigm that includes simpler sorts such as insertion and selection. While some of the rules are specific to LISP, about three-fourths of the rules are independent of LISP or any other target language.

Internally, PECOS's rules are represented as condition-action pairs. The conditions are particular configurations of abstract operations and data structures that are matched against parts of the developing program. Where the match is successful, the actions replace parts of the abstract concepts with refinements of those parts.

In the system of refinement rules, intermediate-level abstractions play a major role. One benefit of such intermediate-level concepts is a certain economy of knowledge. Consider, for example, the construct of a *sequential collection*: a linearly ordered group of locations in which the elements of a collection can be stored. Since there is no constraint on how the linear ordering is implemented, the construct can be seen as an abstraction (or generalization) of both linked lists and arrays. Much of what programmers know about linked lists is in common to what they know about arrays, and hence can be represented as one rule set about sequential collections, rather than as two, one about linked lists, and one about arrays. Another benefit of these intermediate-level concepts is that the process of choosing between alternative (valid) rules is facilitated: Attention can be focused on the essential aspects of a choice while ignoring irrelevant details.

Rules about Programming Knowledge

Most currently available sources of programming knowledge (e,g., books and articles) lack the precision required for effective use by a machine. The descriptions are often informal, with details omitted and assumptions unstated. Before this programming knowledge can be made available to machines, it must be made more precise; the assumptions must be made explicit; and the details must be filled in.

PECOS's rules provide much of this precision for the domain of elementary symbolic programming. For example, consider the following rule (an English paraphrase of PECOS's internal representation):

A collection may be represented as a mapping of objects to Boolean values; the default range object is FALSE.

Most programmers know this fact: that a collection may be represented by its *characteristic function*. Without knowing this rule, or something similar, it is almost impossible to understand why a bitstring can be used to represent a set (or, for that matter, why property list markings work). Yet this rule is generally left unstated in discussions of bitstring representations. As another example, consider the following rule:

> *An association table whose keys are integers from a fixed range may be represented as an array subregion.*

The fact that an array is simply a way to represent a mapping of integers to arbitrary values is well known and usually stated explicitly. The detail that the integers must be from a fixed range is usually not stated. Note that if the integers are not from a fixed range, then an array is the wrong representation and something like a hash table should be used.

PECOS's rules also identify particular design decisions involved in programming. For example, one of the crucial decisions in building an enumerator of the objects in a sequential collection is selecting the order in which they should be enumerated. This decision is often made only implicitly. For example, the use of the LISP function MAPC to enumerate the objects in a list assumes implicitly that the stored (or "natural") order is the right order in which to enumerate them. While this is often correct, there are times when some other order is desired. For example, the selector of a selection sort involves enumerating the objects according to a particular ordering relation. A second major decision in building an enumerator involves selecting a way to save the state of the computation between calls to the enumerator. The use of a location (e.g., index or list cell) to specify the current state is based on knowing the following rule:

> *If the enumeration order is the same as the stored order, the state of an enumeration may be represented as a location in the sequential collection.*

Were the enumeration order different from the stored order (as in a selection sort), then some other state-saving scheme would be needed, such as deleting the objects or marking them in some fashion.

Another interesting aspect of PECOS's rules is that they have a certain kind of explanatory power. Consider, for example, a well-known trick for computing the intersection of two linked lists of atoms in linear time: Map down the first list and put a special mark on the property list of each atom; then map down the second list collecting only those atoms whose property lists contain the special mark. This technique can be understood on the basis of the following four of PECOS's rules (in addition to the rules about representing collections as linked lists):

> *A collection may be represented as a mapping of objects to Boolean values; the default range object is FALSE.*

> *A mapping whose domain consists of atoms may be represented using property list markings.*

> *The intersection of two collections may be implemented by enumerating the objects in one, and while enumerating them, collecting those that are members of the other.*

If a collection is input, its representation may be converted into any other representation before further processing.

Given these rules, the trick works by first converting the representation of one collection from a linked list to property list markings with Boolean values, and then computing the intersection in the standard way, except that a membership test for property list markings involves a call to GETPROP rather than a scan down a linked list.

Status

PECOS is able to implement abstract algorithms (i.e., a very high-level specification) in a variety of domains, including elementary symbolic programming (simple classification and concept formation algorithms), sorting (several versions of selection and insertion sort), graph theory (a reachability algorithm), and even simple number theory (a prime number algorithm). In each case, PECOS's knowledge about different implementation techniques enabled the construction of a variety of alternative implementations, often with significantly different efficiency characteristics.

PECOS's success demonstrates the viability of the knowledge-based approach to automatic programming. In order to develop this approach further, two research directions seem particularly useful.

First, programming knowledge for other domains must be codified. In the process, rules developed for one domain may be found to be useful in other domains. With the hope of verifying the wider utility of PECOS's rules about collections and mappings, Yale's Knowledge-based Automatic Programming Project Barstow, 1978 is currently codifying the programming knowledge needed for elementary graph algorithms.

As an example, consider the common technique of representing a graph as an adjacency matrix. In order to construct such a representation, only one rule about graphs need be known:

A graph may be represented as a pair of sets: a set of vertices (whose elements are primitive objects) and a set of edges (whose elements are pairs of vertices).

The rest of the necessary knowledge is concerned with sets and mappings and is independent of its application to graphs. For example, in order to derive the bounds on the matrix, one need only know that primitive objects may be represented as integers, that a set of otherwise unconstrained integers may be represented as a sequence of consecutive integers, and that a sequence of consecutive integers may be represented as lower and upper bounds. To derive the representation of the matrix itself, one need only know PECOS's rules about Boolean mappings and association tables, plus the fact that a table whose keys are pairs of integers in fixed ranges may be represented as a two-dimensional matrix.

Second, different types of programming knowledge need to be codified. Two types seem particularly important: efficiency knowledge and strategic knowledge. LIBRA (article D9), which acts together with PECOS in PSI's synthesis phase, embodies a large amount of efficiency knowledge; but much remains to be done. Very little work on the use of general strategies (e.g., divide and conquer) in program synthesis has been done. The latter seems an especially important direction, since such strategies seem to play a major role in human programming.

References

See Barstow & Kant (1977), Barstow (1978), Barstow (1979), Green (1976b), Kant (1977), and Teitelman et al. (1978).

G. DEDALUS

DEDALUS, the DEDuctive ALgorithm Ur-Synthesizer, accepts an unambiguous, logically complete, very high-level specification of a desired program and through repeated application of transformation rules seeks to reduce it to an implementation within a simple LISP-like target language. This target language implementation is guaranteed to be correct (i.e., logically equivalent to the high-level specification) and to terminate. The knowledge that ultimately relates the constructs of the specification language to those in the target language is expressed in the transformation rules. But of special importance are certain rules that express general programming principles that are independent of the particular specification language and target language. These rules, which have constituted a major component of the DEDALUS effort, form conditional statements and recursive and nonrecursive procedures; they also generalize procedures, construct well-founded orderings to guarantee the termination of recursive calls, and write code that simultaneously achieves two or more goals. These general programming principles are described in detail in a subsequent section, with examples illustrating their application. As pointed out in the STATUS section, some of the principles are fairly well understood, while others require further study. Not all the principles are implemented in the current DEDALUS system.

The DEDALUS specification language can contain constructs that are close to how the user actually thinks about the problem. Thus, the DEDLAUS specification of the program **lessall(x l)**, which tests whether a number x is less than every element of a list l of numbers, and the program gcd(x y), which computes the greatest common divisor of two nonnegative integers x and y, are specified as follows:

```
lessall(x l) <= compute x<all(l)
                    where x is a number and l is a list of numbers,
gcd(x y)   <= compute max (z:z|x and z|y)
                 where x and y are nonnegative nonzero integers .
```

The all construct in P(all (l)), indicating that the condition P holds for all elements of the list l, and the set constructor {u: P(u)}, indicating the set of elements for which P is true, are constructs that, through the repeated application of transformation rules will eventually be converted into target language code that, for the particular program, is logically equivalent to the original specification. The specification language is not fixed: New constructs can be introduced by modifying or adding transformation rules.

The operation of DEDALUS consists of the repeated application of transformations to expressions in order to produce expressions that are closer to, or within, the target language. In DEDALUS, the expressions that occur during the transformation process specify not only programs; they can also specify conditions to be proved, as well as conditions to be made true. All these expressions are treated as goals to be achieved: For an expression that specifies a program, the goal is to convert that program into a target language implementation; for an expression that is a condition to be proved, the goal is to convert it to the logical constant true; for an expression that is a condition to be made true, the goal is to construct a program that will make that condition true.

Transforming a subexpression (of an expression) into another subexpression requires rules of the form

$$t \Rightarrow t' \text{ if } P \quad ,$$

the condition P being optional. This rule indicates that the subexpression t can be replaced by t'. If P is present, then the rule can only be applied provided that the system first prove that P is true; which is to say, before the rule can be applied, the system must succeed in achieving the subgoal

Goal: prove P .

For example, consider

P(all(l)) => P(head(l)) and P(all(tail(l))) if not empty (l) ,

which expresses the fact that a property P holds for every element of a nonempty list l if it holds for the first element **head(l)** and for every element of the list **tail (l)** of the other elements. Before the system can apply this rule to some part of an expression, it would have to succeed in proving that l is not empty.

The application of transformation rules results in a tree of goals and subgoals. Initially the top-level goals of this tree are established by program specifications. Thus, the common form of program specification

```
f(x) <=            compute P(x)
                   where Q(x) ,
```

establishes its output description as the top-level goal

Goal:compute P(x) ,

and in trying to achieve this goal, the system assumes the truth of Q(x). If the top-level goals of trees are established by program specifications, most goals are established as the result of transformations. Thus, by applying the transformation rule

u|v and u|w => u|v and u|w-v

to the top-level goal of the gcd program

Goal 1: compute max(z:z|x and z|y),

the system establishes

Goal 2: compute max (z:z|x and z|y-x)

as a subgoal. Such transformations express knowledge about specific constructs. In the DEDALUS system there is also knowledge of a more general sort.

General Programming Principles

This section describes five general programming principles and presents several examples to illustrate their application. The principles express knowledge about how to form conditionals and procedures (recursive and nonrecursive), how to replace two or more procedures by a generalized procedure, and how to achieve simultaneous goals. As explained in the STATUS section, the current implementation of DEDALUS does not incorporate the generalization of procedures or the achievement of simultaneous goals.

Conditional formation. Many of the transformation rules impose some condition P (e.g., l is nonempty, x is nonnegative) that must be satisfied for the rule to be applied. Suppose that in attempting to apply a particular rule, the system failed to prove or disprove the condition P, where P is expressed entirely in terms of the primitive constructs of the target language; in such a situation, the conditional formation rule is invoked. This rule allows the introduction of case analysis to consider separately the cases in which P is true and in which P is false. Suppose the result is both a program segment S1 that achieves the goal under the assumption that P is true and another program segment S2 that achieves the goal under the assumption that P is false. The conditional formation principle puts these two program segments together into a conditional expression

```
                    if P then S1 else S2  ,
```

which solves the problem regardless of whether P is true or false. During the generation of S2, the system could discover that a conditional expression was unnecessary: The generation of S2 may not have required the assumption that P was false. In such a case, the program constructed would be simply S2.

Recursion formation. Suppose, in constructing a program with specifications

```
        f(x) <=          compute P(x)
                         where Q(x)  ,
```

the system encounters a subgoal

```
    compute P(t)  ,
```

which is an instance of the output specification, compute P(x). Because the program f(x) is intended to compute P(x) for any x satisfying its input specification Q(x), the recursion formation rule proposes achieving the subgoal by computing P(t) with a recursive call f(t). For this step to be valid, it must ensure that the input condition Q(t) holds when the proposed recursive call is executed. To ensure that the new recursive call will not cause the program to loop indefinitely, the rule must also establish a termination condition, showing that the argument t is strictly less than the input x in some well-founded ordering. (A well-founded ordering is an ordering in which no infinite strictly decreasing sequences can exist.) This condition precludes the possibility that an infinite sequence of recursive calls occur during the execution of the program.

Example: **lessall**. The DEDALUS system derived the program lessall(x l), which tests whether a given number x is less than every element of a give list l of numbers. The specifications for this program are

```
    lessall(x l) <=      compute x < all (l)
                         where x is a number and l is a list of numbers  .
```

In deriving this program, the system develops a subgoal

```
compute x < all(tail(l)) ,
```

in the case that l is nonempty. This subgoal is an instance of the output specification of the original specification, with the input l replaced by tail(l); therefore, the recursion formation principle proposes that the subgoal be achieved by introducing a recursive call lessall(x tail(l)). To ensure that this step is valid, the rule establishes an input condition that

```
x is a number and tail(l) is a list of numbers ,
```

and a termination condition that the argument pair (x tail(l)) is less than the input pair (x l) in some well-founded ordering. This termination condition holds because tail(l) is a proper sublist of l.

As the final program the system obtains

```
lessall(x l) <= if empty(l) then true
                  else x < head (l) and lessall (x tail(l)) .
```

Procedure formation. Suppose that while developing a tree for a specification of the form

```
f(x) <= compute P(x)
          where Q(x) ,
```

the system encounters a subgoal

```
Goal B:  compute R(t) ,
```

which is an instance not of the output specification compute P(x) but of some previously generated subgoal

```
Goal A:  compute R(x) .
```

Then the procedure formation principle introduces a new procedure, g(x), whose output specification is

```
g(x) <= compute R(x) .
```

In this way, both Goals A and B can be achieved by calls g(x) and g(t) to a single procedure. In the case where Goal B has been derived from Goal A, the call to g(t) will be a recursive call; otherwise, both calls will be simple procedure calls.

Example: cart. The specification of the program cart(s t) to compute the Cartesian product of two sets, s and t, is

```
cart(s t) <= compute {(x y) : x∈s and y∈t}
                where s and t are finite sets .
```

While deriving the tree for the program, the system obtains a subgoal

Goal A: compute ((x y) : x=head(s) and y∈t) ,

given that s is nonempty. Developing Goal A further, the system derives

Goal B: compute ((x y) : x=head(s) and y∈tail(t)) ,

given that t is nonempty. Goal B is an instance of Goal A; therefore, the procedure formation rule proposes introducing a new procedure **carthead (s t)** whose output specification is

```
carthead(s t) <= compute ((x y) : x=head(s) and y∈t)
```

so that Goal A can be achieved with a procedure call **carthead(s t)**, and Goal B, with a (recursive) call **carthead(s tail(t))**.

Constructing the carthead procedure by the techniques already described, the final system of programs becomes,

```
cart(s t) <= if empty(s) then ()
             else union( carthead(s t) cart(tail(s) t)) ,


carthead(s t) <= if empty(t) then ()
                 else union( ((head(s) head(t)))
                             carthead(s tail(t))) .
```

Generalization. Suppose, in deriving a program, that we obtain two subgoals

Goal A: compute R(a(x))

Goal B: compute R(b(x)) ,

neither of which is an instance of the other, but both of which are instances of the more general expression

compute R(y) .

In such a case the extended procedure formation rule proposes the introduction of the new procedure, whose output specification is

```
g(y) <= compute R(y) .
```

Thus, Goal A and Goal B can be achieved by procedure calls to g(a(x)) and g(b(x)), respectively.

Example: **reverse**. In constructing a program reverse (l), to reverse a list l, we first derive two subgoals:

Goal A: compute append(reverse(tail(l))

```
                    cons(head(l)nil))


        Goal B:  compute append(reverse(tail (tail(l)))
                          cons(head(tail(l))
                             cons(head(l) nil)))  .
```

skip

Each is an instance of the more general expression

```
        compute append(reverse(tail(l))
                 cons(head(l) m))  ;
```

therefore, the extended procedure formation rule proposes introducing a new procedure **reversegen(l m)**, whose output specification is the more general expression:

```
        reversegen(l m) <= compute append(reverse(tail(l))
                                 cons(head(l) m))  .
```

Although this procedure, which reverses a nonempty list l and appends the result to m, is a more general problem than the original **reverse** program, it turns out that **reversegen** is actually easier to construct. The final system of programs obtained is

```
        reverse(l) <= if empty(l) then nil
                 else reversegen(l nil)

        reversegen(l m) <= if empty(tail(l))   then cons(head(l) m)
                   else reversegen(tail(l)  cons(head(l) m))  .
```

Simultaneous goals. In order to deal with operations that produce side-effects such as modifying the structure of data objects (e.g., assignment statements), DEDALUS introduces constructs such as **achieve** P, to denote a program intended to make the condition P true.

In constructing a program to achieve two conditions, P1 and P2, it is not sufficient to decompose the problem by constructing two independent programs to achieve P1 and P2, respectively. The concatenation of the two programs might not achieve both conditions because the program that achieves P2 may in the process make P1 false, and vice versa.

For example, suppose a program is desired to sort the values of three variables x, y, and z; in other words, to permute the values of the variables to achieve the two conditions $x \geq y$ and $y \geq z$ simultaneously. Assume the given primitive instruction sort2(u v), which sorts the values of its input variables u and v. The concatenation

```
        sort2(x y)
        sort2(y z)
```

of these two segments will not achieve both conditions simultaneously; the second segment sort2(y z) may, by sorting y and z, make the first condition $x \geq y$ false.

The simultaneous goal principle, which was introduced to circumvent such difficulties, states that to satisfy a goal of form

achieve P1 and P2 ,

first construct a program F to achieve P1, then modify F to achieve P2 while protecting P1 at the end of F. A special "protection mechanism" (cf. (Sussman, 1975)) ensures that no modification is permitted that destroys the truth of the protected condition P1 at the end of the program.

Example: sort. To apply this principle to the goal

achieve $x < y$ and $y < z$

in the sorting problem, a system would first achieve $x < y$, by using the segment sort 2(x y). This program would then be modified to achieve the second condition $y \le z$. But adding sort2(y z) at the end of the program will not work because it destroys the truth of the protected condition $x \le y$.

However, in general, a goal may be achieved by inserting modifications at any point in the program, not merely at the end. Introducing the two instructions

```
if y < x then sort2(x y)

if x  y then sort2(y z)
```

at the beginning of the program segment would simultaneously achieve both conditions x y and y z. The resulting program would be

```
if y < x then sort2(x z)
if x < y then sort2(y z)
sort2(x y) .
```

Status

Currently, the DEDALUS implementation incorporates the principles of conditional formation, recursion formation (including the termination proofs), and procedure formation, but it does not include generalization or the formation of structure-changing programs. The techniques for deriving straight-line structure-changing programs were implemented in a separate system (see Waldinger, 1977).

Conditional formation and recursion formation are well understood. The method for proving termination of ordinary recursive calls does not always extend to the multiple-procedure case. The generalization mechanism and the extended procedure formation principle are just beginning to be formulated.

The derivation of straight-line programs with simple side-effects is fairly well understood, but much work needs to be done on the derivation of structure-changing programs with conditional expressions and loops, as well as on the derivation of programs that alter list structures and other complex data objects.

The DEDALUS system is implemented in QLISP (Wilber, 1976), an extension of INTERLISP (Teitelman et al., 1978) that includes pattern-matching and backtracking facilities. The full power of the QLISP language is available in expressing each rule since the rules are represented as QLISP programs in a fairly direct manner.

To date, these are some of the representative samples of the programs constructed by the current DEDALUS system:

Numerical Programs:

- the subtractive gcd algorithm,
- the Euclidean gcd algorithm,
- the binary gcd algorithm, and
- the remainder of dividing two integers.

List Programs:

- finding the maximum element of a list,
- testing if a list is sorted,
- testing if a number is less than every element of a list of numbers (lessall)), and
- testing if every element of one list of numbers is less than every element of another.

Set Programs:

- computing the union or intersection of two sets,
- testing if an element belongs to a set,
- testing if one set is a subset of another, and
- computing the cartesian product of two sets (cart).

References

See Balzer (1972), Balzer, Goldman, & Wile (1977b), Boyer & Moore (1975), Buchanan & Luckham (1974), Burstall & Darlington (1977), Dijkstra (1975), Dijkstra (1976), Green (1976b), Guttag, Horowitz, & Musser (1976), Heidorn (1976), Manna & Waldinger (1978), Siklossy (1974), Sussman (1975), Teitelman et al. (1978), Waldinger (1977), Warren (1974), Warren (1976), and Wilber (1976).

H. PROTOSYSTEM I

PROTOSYSTEM I, an automatic programming system designed by William Martin, Gregory Ruth, Robert Baron, Matthew Morgenstern, and others of the MIT Laboratory for Computer Science, is part of a larger research project aimed at modeling, understanding, and automating the writing of a data-processing system. Hereafter the data-processing system is referred to as a *data-processing program*, in accord with this chapter's terminology, which refers to the output of an automatic programming system as a program. A model of the larger research project was developed that consists of five phases. The successive phases can be viewed as a series of transformations of the descriptions of the target program, beginning with a global conceptual description of the problem at hand and progressing, through increasing specificity, toward a detailed machine-level solution. The aim of the project is to develop stages of an automatic programming system where each corresponds to one of the five phases of the model and each embodies the particular knowledge and expertise for that phase.

Phase 1: Problem Definition--The specification of the data-processing program is expressed in domain-dependent terms in English.

Phase 2: Specification Analysis and System Formulation--The specification in Phase 1 is viewed as a data-processing problem. This problem is solved, yielding a data-processing formulation of the desired program.

Phase 3: Implementation--The procedural steps, data representation, and organization of the target are determined by intelligent selection from, and adaptation of, a set of standard implementation possibilities.

Phase 4: Code Generation--The implementation of Phase 3 is transformed into code in some high-level language (e.g., PL/I).

Phase 5: Compilation and Loading--The high-level code is transformed into a form that can be "understood" and executed by the target computer.

The first two phases involve such AI areas as natural language comprehension, program model formation, and problem solving. Since these areas are still in the process of evolution, the development of the first two phases has been deferred. At present, PROTOSYSTEM is limited to the automation of phases 3 and 4 since it was felt that these phases were much more amenable to solution. Thus, the current PROTOSYSTEM accepts a specification in terms of abstract relations (in a very high-level language called SSL), and then designs an optimized data-processing program and generates code for an efficient implementation. In automatic programming it is usually impossible for a system to carry out a search for the absolutely optimal implementation; instead, a system works at optimizing a program only to a degree.

The particular problem area of PROTOSYSTEM I is that of I/O intensive (file manipulation and updating), batch-oriented, data-processing programs. Included in this area are programs for inventory control, payroll, and other record-keeping systems.

The specification method uses a description of the desired data-processing program in the SSL language. An SSL specification consists of a data and a computation division. The data division gives the names of data sets (conceptual aggregations or groupings of data), their keys, and their period of updating. The computation division specifies for each computed file the calculations to be performed when it is computed. Figure 1 illustrates an SSL specification of a data-processing program for a warehouse inventory. In the proposed problem, the warehouse stocks a number of different kinds of items that are sent out daily to various stores. The data-processing program's task is to keep track of inventory levels, which items and how many of each item should be reordered from the producer (an item is reordered when less than 100 remain in stock), and how many items are received from the producer. In the data division are data sets (e.g., shipments-received, beginning-inventory, total-items, etc.), and in the computation division are the computation steps that involve these data sets (e.g., for each item, the beginning inventory is computed by adding the shipments received to the final inventory from the previous day).

After receiving the SSL specification of the desired program, PROTOSYSTEM transforms it into an efficient target language implementation consisting of a collection of PL/I programs and its JCL ("Job Control Language") for the IBM 360 system. To accomplish this transformation, the following specific design decisions are made with the goal of achieving an efficient implementation:

(a) Design each keyed file, deciding what are to be its data items, organization (consecutive, index sequential, regional), storage device, associated sort ordering, and number of records per block;

(b) design each job step, determining which computations the step is to include, its accessing method (sequential, random, core table), its driving data set(s), and the order (by key values) in which the records of its input data sets are to be processed;

(c) determine whether sorts are necessary and where they should be performed; and

(d) determine the sequence of job steps.

Generally, these design decisions, especially the central ones of determining the final target data sets, computation steps, and sequencing of computation steps, are made by exploring the different ways of combining data sets and computation steps. The system carries out these explorations with the goal of minimizing the number of file accesses made during the run-time of the target implementation. Sometimes, as explained below, the system also will seek to minimize a more detailed cost estimate of the target implementation.

Described in greater detail in the next section, the method employed by PROTOSYSTEM for achieving an efficient implementation does not rely solely on heuristics but instead uses what is essentially a dynamic programming algorithm with heuristics added to the algorithm, so that it can finish in a reasonable amount of time. An advantage of dynamic programming is that it can provide a good handle on global optimization when the results of individual decisions have far-reaching and compounding effects throughout the design of the data-processing program.

Operation

Although the actual optimization process is performed by the optimizer module, several other modules provide preparatory and support services. First, the structural analyzer module generates predicates for the operations in the SSL computation division. These predicates indicate the conditions under which data items in a data set will be either accessed or generated during an operation. For example, the condition

(DEFINED A (k1)) = (OR (DEFINED B (k1)) (DEFINED C (k1)))

would indicate that there is a record in data set A for a value of the key, k1, only when at least one of the data sets B or C has a record for that value of the key. The structural analyzer also produces candidate driving data sets for each operation in the computation division. A driving data set of an operation is a data set whose records are "walked through" once in order of their occurrence--i.e., the operation is executed once at each step (record)--to drive the operation.

The predicates produced by the structural analyzer are then used by the question-answering module to provide information to the optimizer about the average number of I/O accesses implied by tentative configurations (i.e. tentative choices for the data sets and computation steps) of the target implementation. The question-answering module maintains a knowledge base consisting of the predicates, characteristics of the data, as well as information obtained from interaction with the user, such as average data set size or the probability of a predicate fragment being true. This knowledge, along with knowledge about the probability calculus, is used to answer questions about the size of a data set and about the average number of items in the data set that are likely to satisfy a certain predicate (e.g., an access predicate). When the knowledge is insufficient to answer an optimizer question, the question answerer initiates a dialogue with the user in order to elicit enough additional information to proceed.

The optimization process itself is performed by the optimizer module. This module intermittently obtains information from the question answerer about I/O accesses of tentative configurations of parts of the data-processing program, in order to explore the effects of such design parameters as the number of records per block, the file organization, the data items that are collected into a single data set, and the computations that are performed during a single reading of a file or files. Since the problem area of PROTOSYSTEM is that of I/O intensive programs, the optimizer explores the various design parameters with the goal of minimizing the number of file accesses of the target language implementation (of the data-processing program). Sometimes, however, after a number of more important design decisions have been made, the optimizer will explore design decisions by computing a more detailed cost estimate that attempts to approximate the charging structure of the particular installation on which the target system is to run (e.g., disc space, core residency charges, explicit I/O, etc.).

The central part of the optimization process is concerned with the the exploration of various ways of setting up data sets and computation steps. Basically, the optimization module starts with the data sets and computation steps in the data division and computation division of the SSL specification. Then, with the goal of minimizing the number of file accesses, the module looks at data-processing programs that use various aggregations of these initial data sets and computation steps (an aggregation of two or more data sets is a data set that has all the data items of the original data sets, while an aggregation of several computation steps is a computation step that performs the functions of the original steps). The optimizer explores aggregating data sets and aggregating computation steps and develops and utilizes constraints on the sort order of both data sets and computation steps (an example of a sort order constraint on a data set would be when the data set should have its records sorted on a particular key first).

To avoid the problem of combinatorial explosion, the module uses a form of dynamic programming with heuristics. Loosely speaking, one may say that dynamic programming is a set of parameterized recursive equations, which, in this case, express the cost of optimized longer segments of the program in terms of optimized shorter segments. A pure dynamic programming algorithm, though it would find the absolute optimum target implementation, would require an extreme amount of time to do so. Therefore, in order that the algorithm finish in a reasonable time, a number of heuristics have been employed in the algorithm, including decoupling decisions where possible (and sometimes even where it is not completely possible) and carrying out local optimizations before making adjustments for global concerns. A full explanation of the algorithm is found in Morgenstern (1976).

Status

The SSL specification language has been completely defined and there is an operational implementation of PROTOSYSTEM in MACLISP on the MIT LCS PDP-10. The system is capable of producing acceptable target language implementations. From a larger perspective, the PROTOSYSTEM I project has developed a 5-phase model of the process of writing a data-processing program (system), from its conception to its implementation as executable code. Twenty years ago, the fifth phase, compilation and loading, was automated. At present, a preliminary theory and automation of the third and fourth phases, the generation of the system and translation into high-level code, are embodied in PROTOSYSTEM I. It is felt that within the next decade the theory and automation of the remaining two phases, including problem definition, specification analysis, and system formulation, should easily fall within the realm of presently developing AI technologies.

DATA DIVISION

```
FILE SHIPMENTS-RECEIVED                FILE QUANTITY-ORDERED-BY-STORE
   KEY IS ITEM                            KEY IS ITEM
   GENERATED EVERY DAY                    GENERATED EVERY DAY
FILE BEGINNING-INVENTORY               FILE TOTAL-SHIPPED
   KEY IS ITEM                            KEY IS ITEM, STORE
   GENERATED EVERY DAY                    GENERATED EVERY DAY
FILE TOTAL-ITEM-ORDERS                 FILE FINAL-INVENTORY
   KEY IS ITEM                            KEY IS ITEM
   GENERATED EVERY DAY                    GENERATED EVERY DAY
FILE QUANTITY-SHIPPED-TO-STORE         FILE REORDER-AMOUNT
   KEY IS ITEM, STORE                     KEY IS ITEM
   GENERATED EVERY DAY                    GENERATED EVERY DAY
```

```
                         COMPUTATION DIVISION

BEGINNING-INVENTORY IS
     FINAL-INVENTORY (from the previous day) + SHIPMENTS-RECEIVED
TOTAL-ITEM-ORDERS IS SUM OF QUANTITY-ORDERED-BY-STORE FOR EACH ITEM
QUANTITY-SHIPPED-TO-STORE IS
     QUANTITY-ORDERED-BY-STORE          IF BEGINNING-INVENTORY IS
                                   GREATER THEN TOTAL-ITEM-ORDERS
 ELSE
     QUANTITY-ORDERED-BY-STORE
     * (BEGINNING-INVENTORY / TOTAL-ITEM-ORDERS)
                                   IF BEGINNING-INVENTORY IS NOT
                                   GREATER THEN TOTAL-ITEM-ORDERS
TOTAL-SHIPPED IS SUM OF QUANTITY-SHIPPED-TO-STORE FOR EACH ITEM
FINAL-INVENTORY IS BEGINNING-INVENTORY - TOTAL-SHIPPED
REORDER-AMOUNT IS 1000 IF FINAL-INVENTORY IS LESS THAN 100.
```

Figure 1: SSL relational description for a data processing program.

References

See Baron (1977), Morgenstern (1976), Ruth (1976a), Ruth (1978), and Ruth (1979).

I. NLPQ: Natural Language Programming for Queuing Simulations

The Natural Language Programming for Queuing Simulations (NLPQ) project was begun by George Heidorn at Yale University in 1967 as a doctoral dissertation and completed at the Naval Postgraduate School during the years 1968-1972. The problem area is that of simulation programs for simple queuing problems. The queuing problem's specification occurs during an English dialogue in which the user and the NLPQ system each can furnish information to, and request information from, the other. From this dialogue, the NLPQ system creates and maintains a partial internal description of the queuing problem. This partial description is used to answer any questions that the user may ask; it is used to generate questions that are to be asked of the user; and when eventually completed by the dialogue activity, it is used to generate the implementation of the simulation program in the target language GPSS. The system's processing -- including creating the problem description and generating the GPSS program, as well as translating and generating sentences--is specified by production rules.

Specification

In the English dialogue that constitutes NLPQ's method of specification, the user can make statements, give commands, ask questions, and answer questions. During this process the system can ask and answer questions and respond to commands. In the very brief dialogue of Figure 1, most of the features of the specification method are illustrated in a simple way (other more complex dialogues are presented in Heidorn, 1972, Heidorn, 1974, Heidorn, 1975b, and Heidorn, 1976). In the dialogue, line numbers have been added for purposes of reference; and the lower case typing was done by the user, while the UPPER CASE typing was done by the computer.

The particular specification concerns a queuing problem about cars that arrive at a station, get serviced, and leave. The user initially volunteered some information about how often the cars arrive and about the sequencing of actions, in lines 1 and 2, and then asked the system what other information was needed to complete the queuing model, in line 3. In lines 4 through 9, the system asked questions about times and actions, which the user answered, until the problem statement was complete, in line 10. Then in line 11 the user asked a specific question about the queuing model concerning the mean time needed to service a car, a question that the system answered in line 12. In line 13 the user told the system to produce a description of the entire problem in English for verification purposes, which it did, in line 14. Finally, in response to the user command in line 15, the system produced the GPSS program, in line 16, complete with English comments (preceded by *) and meaningful symbolic names. The time required for the NLPQ system to process this sample specification and to produce the GPSS program was 8 seconds of virtual CPU time on an IBM 370/168 running LISP.

1 ** EVERY 6 MINUTES A CAR ARRIVES AT A STATION.
2 ** AFTER ARRIVING, IT IS SERVICED AT A PUMP.
3 ** IS THE PROBLEM STATEMENT COMPLETE?
4 How long are the cars serviced at the pump?
5 ** FROM 3 TO 7 MINUTES.
6 After being serviced at the pump, what do the cars do?
7 ** THEY LEAVE THE STATION.
8 How long shall the simulation be run?

9 ** 10 HOURS; USE 1 MINUTE AS THE BASIC TIME UNIT.
10 The problem statement is complete.
11 ** WHAT IS THE MEAN OF THE TIME TO SERVICE CARS?
12 *The mean of the uniform distribution is 5 minutes.*
13 ** DESCRIBE THE PROBLEM IN ENGLISH.
14 The cars arrive at the station every 6 minutes.
After arriving at the station, the cars are serviced at the
pump. The time for the cars to be serviced at the pump is
uniformly distributed, with a mean of 5 minutes and a
half-range of 2 minutes. After being serviced at the pump,
the cars leave the station.
The simulation is to be run for 10 hours, using a
basic time unit of 1 minute.
15 ** WRITE THE GPSS PROGRAM.

```
 SIMULATE
 STAT1 EQU      1,F,Q
 PUMP2 EQU      2,F,Q
 CAR1  EQU      1,T
 1     TABLE    M1,1,1,2
*
*      THE CARS ARRIVE AT THE STATION.
       GENERATE 6
       ASSIGN   1,CAR1
*
*      THE CARS ARE SERVICED AT THE PUMP.
ACT1   QUEUE    PUMP2
       SEIZE    PUMP2
       DEPART   PUMP2
       ADVANCE  5,2
       RELEASE  PUMP2
*
*      THE CARS LEAVE THE STATION.
ACT3   TABULATE P1
       TERMINATE
*
*      TIMING LOOP
       GENERATE  600
       TERMINATE 1
       START     1
       END
```

Figure 1. A very brief NLPQ dialogue.

Operation

The processing to be done by NLPQ is specified by sets of production rules written in a language designed especially for this system. "Decoding" rules specify how strings of

English text are to be converted into records in a semantic net, and "encoding" rules specify how records are to be converted into text. These rules are basically phrase structure grammar rules Natural Language.B1, but they are augmented with arbitrary conditions and structure-building actions .

The representation of the internal description of the simulation problem as well as the representation of the syntactic and semantic structures are in the form of a semantic network Representation.B2. A network consists of records that represent such things as concepts, words, physical entities, and probability distributions. Each record is a list of attribute-value pairs, where the value of an attribute is usually a pointer to another record but may sometimes be simply a number or character string.

Prior to a queuing dialogue, the system is given a network of about 300 "named" records containing information about words and concepts relevant to simple queuing problems. Also, it is furnished with a set of about 300 English decoding rules and 500 English and GPSS encoding rules. As the dialogue progresses, the system uses the information it obtains from the English dialogue to build and complete a partial description of the desired simulation, a description that is in the form of a network called the Internal Problem Description (IPD).

Basically, an IPD network describes the flow of mobile entities, such as vehicles, through a framework consisting of stationary entities, such as pumps, by specifying the actions that take place in the framework and their interrelationships. Each action is represented by a record whose attributes furnish such information as the type of action, the entity doing the action (i.e., the agent), the entity that is the object of the action, the location where it happens, its duration, its frequency of occurrence, and what happens next. For example, the action "The men unload the truck at a dock for two hours" could be represented by the record:

```
R1:    Type      unload
        Agent     men
        Object    truck
        Location  dock
        Duration  2 hours
```

From the English dialogue the NLPQ system must obtain all the information needed to build the IPD. Thus, the user must describe the flow of mobile entities through the queuing model by making statements about the actions that take place and about the relations between these actions. Each mobile entity must "arrive" at or "enter" the model. Then it may go through one or more other actions, such as "service," "load," "unload," and "wait." Then, typically, it "leaves" the model. The order in which these actions take place must eventually be made explicit by the use of subordinate clauses beginning with such conjunctions as "after," "when," and "before," or by using the adverb "then." If the order of the actions depends on the state of the queuing model, an "if" clause may be used to specify the condition for performing an action; a sentence with an "otherwise" in it is used to give an alternative action to be performed when this condition is not met.

The information needed to simulate the problem, including the various times involved, must also be furnished by the English dialogue. It is necessary to specify the time between arrivals, the time required to perform each activity, the length of the simulation run, and the basic time unit to be used in the GPSS program. Inter-event and activity times may be given as constants or as probability distributions, such as uniform, exponential, normal, or empirical. The quantity of each stationary entity should also be specified, unless 1 is to be assumed.

The user may either furnish this information in the form of a complete problem statement or state some part of it and then let the system ask questions to obtain the rest of the information, as was done above in lines 1 through 10 of Figure 1. The latter method results in a scan of the partially built IPD for missing or erroneous information and the generation of appropriate questions. Each time the system asks a question, it is trying to obtain the value of some specific attribute that will be needed to generate a GPSS program. To furnish a value for the attribute, the question may be answered by a complete sentence or simply by a phrase.

The user may ask the system specific questions about the queuing model, and then the system generates the answers from the information in the appropriate parts of the IPD. In order to check the entire IPD as it exists at any time, the user may request that an English problem description be produced. Such a description consists of all the information in the IPD as it is converted into English by the encoding rules (see line 14 of Figure 1). Specifically, for each action in the IPD, the system generates one or more statements describing the type of action, its agent, object, location, what action if any follows (if none, a new paragraph is started), and, if applicable, an inter-event time or duration. Conditional successor actions may result in two sentences, with the first one having an "if" clause in it and the second one beginning with "otherwise." After all of the actions have been described, a separate one-sentence paragraph is produced with the values of the run time and the basic time unit.

After the dialogue is finished and all the required information is obtained, NLPQ uses the IPD and the GPSS encoding rules to produce the desired program in the GPSS target language. Such a program was listed in 16 of Figure 1. At the beginning of this program, the definitions for the stationary entities, mobile entities, and distributions are given. Then, for each action, a comment consisting of a simple English action sentence is produced, followed by the GPSS statements appropriate to this action. For example, an "arrive" usually produces a GENERATE and an ASSIGN; a "leave" produces a TABULATE and a TERMINATE; and most activities produce a sequence like QUEUE, SEIZE, DEPART, ADVANCE, and RELEASE. These are usually followed by some sort of TRANSFER, depending upon the type of value that the action's successor attribute has. Finally, the GPSS program closes with a "timing loop" to govern the length of the simulation run.

Status

Though this project was "completed," a system ready for production use was not developed. The NLPQ prototype, however, was demonstrated several times on a variety of problems. Although the capabilities of the implemented system are limited, the research did establish an overall framework for such a system, and useful techniques were developed. Enough details were worked out to enable the system to carry out interesting interactions, as evidenced by the longer more complicated dialogues found in the first four references at the end of this article. More details of the processing done by this system can be found in any of the references , especially Heidorn, 1972, which is a 376-page technical report.

References

See Heidorn (1972), Heidorn (1974), Heidorn (1975a), Heidorn (1975b), and Heidorn (1976).

J. LIBRA

LIBRA, the efficiency analysis expert of the PSI system (Article D2) is being developed by Elaine Kant in conjunction with the PSI project at Systems Control, Inc., and at Stanford University. The PSI system, through interaction with the user, constructs a very high-level program specification called the program model. Then LIBRA, working together with the PECOS coding expert D5, converts the program model into a target language implementation. The PECOS system supplies the transformation rules that can convert the program model into various target language implementations. Using global efficiency analysis ("global analysis" is analysis with access to the entire program, as opposed to only a local segment), LIBRA directs and explores the application of the transformation rules so as to produce an efficient implementation.

The transformation process itself consists of repeated applications of transformation rules to parts of the program, where every application results in a specification closer to a target language implementation. Each such application of a rule is said to produce a partial implementation or *refinement* of the program, and the transformation rules are called *refinement rules*. Thus refinement rules applied to refinements produce further refinements. Because more than one refinement rule may be applicable to the same part of a refinement, the transformation process produces a tree of possible refinements (the actual situation is slightly more complicated since the order in which the rules are applied can affect the tree that is produced). To avoid the problem of combinatorial explosion, LIBRA develops only part of the tree. A discussion of the details of this process follows.

It is LIBRA's function to analyze and guide the development of the refinement tree in order to achieve an efficient implementation. LIBRA determines what parts of the program to expand next and what parts not to expand at all. In particular, when more than one refinement rule is applicable, LIBRA may decide to apply them all so that the resulting refinements can be considered in greater detail; or LIBRA may decide to apply only one of the rules. In the latter case, the refinement is implemented directly in the current node of the tree, and the other possibilities are permanently forgone.

One of the most important ways in which LIBRA attacks the problem of combinatorial explosion is by *estimating* the efficiency of possible target language implementations. For each refinement in the tree, LIBRA maintains two cost estimates; the estimates are in the form of symbolic algebraic expressions that give the time and space requirements needed to execute a certain kind of target language implementation. The first estimate is the default cost that might result if all the constructs and operators in the refinement were assigned default implementations. The second is the optimistic cost estimate that might result assuming: (a) certain efficient implementation techniques that have worked in similar situations will prove succesful in the present situation, and (b) LIBRA expends enough of its own resources of time and space to carry out these implementation techniques.

Treating these two costs as upper and lower bounds on the costs of possible target language implementations of the refinement, LIBRA obtains important guidance in directing the growth of the refinement tree. These upper and lower bounds can be used to prune a branch of the refinement tree (without further consideration of the branch) or to calculate the effect of a partial implementation decision on the global program cost. As discussed below in the RULES section, the upper and lower bounds are used to direct attention to high impact areas, those areas where effort is likely to yield the greatest increases in overall efficiency.

Another feature of the LIBRA system, a feature implicit in the above, is the knowledge LIBRA has about the use and limits of its own resources of available time and space. This feature is important because no system can devote unlimited effort to finding an efficient implementation. Effort must be allocated. The way in which LIBRA performs this allocation is to assign available resources to high impact areas, where the resources will do the most good. The RULES section will present the method used to compute impact, as well as examples and uses of resource knowledge.

LIBRA also includes mechanisms to assist in the acquisition of new programming concepts. When new high-level constructs are added (such as new types of sorts, or trees), new efficiency knowledge is needed to analyze these concepts (their subparts, running times, data structure accesses, and so on). LIBRA has a model of programming concepts that is consulted when new concepts are added. Some of the necessary information can be deduced automatically, and the user is asked specific questions to obtain the rest. To help construct these estimation functions, LIBRA provides a semi-automatic procedure for deriving cost estimation functions from the set of cost functions for the target language constructs.

The knowledge for managing resources, computing upper and lower cost estimates, directing attention to different parts of the tree, making implementation decisions, and, in general, for analyzing and directing the growth of the tree is in the form of rules. Each rule consists of a condition and an action to be performed if the condition is met. The knowledge that a rule expresses can easily be modified since the rules are replaceable and can be added, deleted, or altered without requiring a modification to the system itself.

Rules

The rules in LIBRA's knowledge base generally can be divided into three groups: attention and resource management rules, plausible-implementation rules, and cost-analysis rules.

Attention and resource management rules describe when to shift attention to other nodes in the tree and also how to set priorities for refining the different constructs and operations within a refinement node. Some of the more important of these rules determine how LIBRA's own resources of available time and space are to be allocated, on the basis of where they will have the greatest impact. One of the ways of determining impact is to consider the difference between the upper bound cost estimate (assuming default implementations) and the optimistic lower bound cost estimate (assuming both the successful application of efficiency techniques that have worked in similar situations and the sufficient expenditure of resources to carry the techniques to completion). Other rules in this group state how to shift attention among nodes. These rules (a) cause complex programs to be expanded early in order to see what decisions are involved, (b) postpone trivial decisions until important ones are made, (c) look at all refinements in the tree and select for development the one whose optimistic cost estimate is least (when resources for developing a particular refinement are exhausted), and (d) apply a form of branch and bound which states that (when resources allocated for considering a particular decision are exhausted) attention should be directed to the whole tree and that all nodes whose optimistic cost estimate is worse than the default estimate of some other node should be eliminated. As described later, when cost analysis rules compare estimates, they take into account the degree of uncertainty in the estimate.

Plausible implementation rules express heuristics about when to limit expansion of nodes, by *making a decision about some part of an implementation*. For example, when the question of how to represent a set first arises, LIBRA performs a global examination of the program to determine all uses of the set. If there are many places where the program checks for membership in the set, then a hash-table representation may be suggested. In general, plausible implementation rules express knowledge derived by human or machine analysis of commonly occurring situations, such as which sorting techniques are best for different size inputs. These rules also contain heuristics to make quick decisions. Thus, if LIBRA is running out of resources, heuristics that are not as dependable as the one just described are used to make decisions on the spot, without creating any new nodes. These heuristics generally express defaults, such as "use lists rather than arrays if the target language is LISP"; they are used to make the less important decisions or to make all decisions if the total resources for writing a program are nearly exhausted.

The final group, the cost-analysis rules, express how to compute, update, and compare upper and lower bound estimates of the cost of the final implementation. The cost estimates are in the form of symbolic algebraic expressions that may involve variables representing set sizes. The cost estimates are not computed once and for all: Whenever a refinement in the tree is further refined (i.e., a refinement rule is applied to some part of a node in the subtree whose root is the refinement), then the cost estimates associated with the refinement are *incrementally updated* so as to produce estimates that are more accurate in view of the new information. Cost estimates are constructed from a knowledge base that includes information on upper and lower bounds on costs for time and space usage by individual constructs and operations, and on how to combine such cost estimates for composite programs. The knowledge needed to incrementally update the cost estimates is contained in rules corresponding to the particular construct or operation. The method of comparing the cost estimates of different refinements involves the addition of a bonus to the refinement that has a greater degree of completion and that consequently has a greater certainty in its cost estimates (default and optimistic). This feature favors a nearly complete refinement that has a slightly worse lower bound over a less complete (more abstract) refinement that has a slightly better lower bound. Such a preference is desirable since the cost estimate of the more abstract refinement is less certain and therefore may not be achievable. By giving a bonus for the degree of completion, the cost analysis rules take into account the likelihood of being able to achieve the cost estimate.

Status

LIBRA has guided the application of the PECOS refinement rules to produce efficient implementation of several variants of simple database retrieval, sorting, and concept formation programs (see PSI article for an example of a concept formation program). Current plans include extending the problem area to include simple algorithms for finding prime numbers and for reaching nodes in a graph. For an efficiency expert to be of use in a complete automatic programming system, a good deal more research is needed. Higher level optimizations, extended symbolic analysis and comparison capabilities, and more domain expertise are some obvious extensions. Automatic bookkeeping of heuristics and perhaps even automatic generation of heuristics from an analysis of symbolic cost estimates of target language concepts are some long-range goals. In order to write more complex programs such as compilers or operating systems, more efficiency rules would have to be added to the system, rules about concepts such as bit-packing, machine interrupts, and multiprocessing.

However, even with such additions, the efficiency techniques employed by the LIBRA system should be significant in controlling the problem of combinatorial explosion that occurs during the search for efficient implementations.

This article closes with the description of an example illustrating LIBRA's present operation producing a simple sort program.

Example

Suppose that a SORT is specified as a transfer of elements from a SOURCE sequential collection to a TARGET sequential collection that is ordered by some relation such as LESS-THAN. After the application of some preliminary refinement rules that do not require any decisions as to alternative choices, three choice points remain: choosing a transfer order, and choosing representations for SOURCE and for TARGET.

Since the transfer order is selected as the most important decision, LIBRA directs attention first to that choice point. A heuristic rule is applied that suggests the use of either an insertion sort from list to list or array to array, or a selection sort from list to array. The different refinement possibilities are added to the tree accordingly. Each of the branches is given a limited amount of resources and told to focus attention only on the parts of the program directly relevant to the transfer order decision.

After these branches are refined within the limits of the assigned resources, the nodes of the tree are compared. Branch and bound does not eliminate any of the aternatives here, but the insertion branch is selected as it has the best lower bound (taking into account factors related to uncertainty of estimates).

Refinement then proceeds in that node. The choice of a list or array representation for the TARGET is made by a heuristic that says that lists are easier to manipulate than arrays in LISP. This heuristic was applied because much of the time and space resources allocated for finding an implementation had been consumed in the above tasks and a quick decision was required. The choice of a list representation for TARGET forces a list representation for SOURCE because of a suggestion made under the transfer-order heuristic. Thereafter, the refinement process is basically straightforward, though several choices of whether to store or recompute local variables are made.

References

See Barstow (1979), Barstow & Kant (1977), Green (1976b), Green (1977), Green & Barstow (1978), Kant (1977), Kant (1978), Kant (1979), and McCune (1977).

References

Automatic Coding. **Proc. of the Symposium**, Franklin Institute, Philadelphia, PA, January 1957.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. **The Design and Analysis of Computer Algorithms**. Reading, Mass.: Addison-Wesley, 1974.

Allen, F. E., & Cocke, J. A catalogue of optimizing transformations. In R. Rustin (Ed.), **Design and Optimization of Compilers, Proceedings of the Courant Computer Science Symposium 5**. Englewood Cliffs, N.J.: Prentice-Hall, 1972. Pp. 1-30.

Allen, F. E., & Cocke, J. A program data flow analysis procedure. **Communications of the ACM**, 1976, **19**(3), 137-147.

Amarel, S. Representation and modeling in problems of program formation. In B. Meltzer & D. Michie (Eds.), **Machine Intelligence 6**. New York: American Elsevier, 1972. Pp. 411-466.

Balzer, R. M. Dataless programming. **Proceedings FJCC**, 1967, **31**, 535-544.

Balzer, R. M. **Automatic Programming**. Information Sciences Institute Tech. Memo 1, University of Southern California, Marina Del Rey, 1972.

Balzer, R. M. CASAP: A testbed for program flexibility. **IJCAI 3**, 1973, 601-605. (a)

Balzer, R. M. A global view of automatic programming. **IJCAI 3**, 1973, 494-499. (b)

Balzer, R. M. **A Language-independent programmer's interface**. Information Sciences Institute Report RR-73-15, University of Southern California, Marina Del Rey, November 1973. (c)

Balzer, R. M. **Human Use of World Knowledge**. Information Sciences Institute Report USC-ISI RR-73-07, University of Southern California, Marina Del Rey, March 1974 (ARPA Order 2223/1).

Balzer, R. M. , & Goldman, N. Principles of Good Software Specification and Their Implications for Specification Languages. **Proc. of the IEEE Specifications of Reliable Software Conf.**, Cambridge, April 1979.

Balzer, R. M., Goldman, N., & Wile, D. On the Transformational Implementation Approach to Programming. **2nd Int. Conf. on Software Engineering**, October 1976, pp. 337-344.

Balzer, R. M., Goldman, N., & Wile, D. Informality in program specification. **IJCAI 5**, 1977, 389-397. (a)

Balzer, R. M., Goldman, N., & Wile, D. Meta-evaluation as a tool for program understanding. **IJCAI 5**, 1977, 398-403. (b)

Balzer, R. M., Goldman, N., & Wile, D. On the Use of Programming Knowledge to Understand Informal Process Descriptions. **SIGART Newsletter**, No. 63, June 1977, pp. 72-75. (c).

Balzer, R. M., Goldman, N., & Wile, D. Informality in Program Specifications. **IEEE Transactions on Software Engineering**, 1978, SE-4(2), 94-103.

Balzer, R. M., Greenfeld, N., Kay, M., Mann, W., Ryder, W., Wilczynski, D., & Zobrist, A. Domain independent automatic programming. IFIP, 1974, 326-330.

Baron, R. V. **Structural Analysis in a Very High Level Language**, Master's thesis, MIT, 1977.

Barstow, D. A Knowledge based system for automatic program construction. IJCAI 5, 1977, 382-388. (a)

Barstow, D. A knowledge base organization for rules about programming. **Proc. of the Workshop on Pattern Directed Inference Systems**. SIGART Newsletter, No. 63, June 1977, pp. 18-22. (b)

Barstow, D. **Automatic Construction of Algorithms and Data Structures using a Knowledge Base of Programming Rules**, AI Memo 308, Computer Science Dept., Stanford University, November 1977. (c)

Barstow, D. **Codification of programming knowledge: Graph algorithms**, TR-149, Computer Science Dept., Yale University, December 1978.

Barstow, D. **Knowledge-based Program Construction**. Elsevier: North Holland, 1979.

Barstow, D. R., & Kant, E. Observations on the interaction between coding and efficiency knowledge in the PSI system. **Proc. of the 2nd Int. Conf. on Software Engineering**, Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, CA, October 1977, pp. 19-31.

Barth, J. M. An interprocedural data flow analysis algorithm. **Fourth ACM Symposium on Principles of Programming Languages**, Los Angeles, CA, January 1977.

Bauer, M. A basis for the acquisition of procedures from protocols. IJCAI 4, 1975, 226-231.

Biermann, A. W. Computer program synthesis from computation traces. **Symposium on Fundamental Theory of Programming**, Kyoto University, Kyoto, Japan, October 1972. (a)

Biermann, A. W. On the inference of Turing machines from sample computations. **Artificial Intelligence**, 1972, 3, 181-198. (b)

Biermann, A. W. The Use of Examples in Program Construction and Debugging. **ACM '75: Proceedings of the National Conference, Association for Computing Machinery**, New York, 1975. Pp. 242-247.

Biermann, A. W. Approaches to automatic programming. In M. Rubinoff & M. C. Yovits (Eds.), Advances in Computers (vol. 15). New York: Academic Press, 1976. Pp. 1-63. (a)

Biermann, A. W. Regular LISP Programs and Their Automatic Synthesis from Examples, CS-1976-12, Dept. of Computer Science, Duke University, June 1976. (b)

Biermann, A. W., Baum, R. I., & Petry, F. E. Speeding up the synthesis of programs from traces. IEEE Transactionson computers, February 1975, C-24, 122-136.

Biermann, A. W., & Krishnaswamy Constructing programs from example computations, OSU CISRC TR-74-5, August 1974.

Biermann, A. W., & Feldman, J. A. On the Synthesis of Finite-state Acceptors, AI Memo 114, AI Lab, Stanford University, April 1970.

Biggerstaff, T. J. C2: A Super-compiler Approach to Automatic Programming. Doctoral dissertation, Tech. Rep. 76-01-01, Dept. of Computer Science, University of Washington, 1976.

Bobrow, D. G., & Wegbreit, B. A model of control structures for Artificial Intelligence programming languages. IEEE Transactions on Computers, 1976, C-25(4), 347-353.

Bobrow, D. G., & Winograd, T. An overview of KRL, a knowledge representation language. Cognitive Science, 1977, 1(1), 3-46.

Boyer, R. S., & Moore, J. S. Proving theorems about LISP Functions. JACM, 1975, 22(1), 129-144.

Brown, G. P. A Framework for Processing Dialogue, TR-182, Laboratory for Computer Science, MIT, June 1977.

Brown, R. Use of Analogy to Achieve New Expertise, AI-TR-403, MIT AI Lab, April 1977.

Buchanan, J. R., & Luckham, D. C. On Automating the Construction of Programs, TR-CS-433, Artificial Intelligence Laboratory, Stanford University, Stanford, CA, May 1974. (Also Stanford AI Memo 236)

Burstall, R. M., & Darlington, J. Some transformations for developing recursive programs. International Conference on Reliable Software, IEEE Computer Society, April 1975, pp. 465-472.

Burstall, R. M., & Darlington, J. A Transformation System for Developing Recursive Programs. Journal of the Association for Computing Machinery, 1977, 24(1), 44-67.

Chandrasekaran, B. AI--The past decade--Automatic programming. In M. Rubinoff & M. C. Yovits (Eds.), Advances in Computers (vol. 13). New York: Academic Press, 1975. Pp. 170-232.

Chang, C., & Lee, R. Symbolic Logic and Mechanical Theorem Proving. New York: Academic Press, 1969.

Cheatham, T. E., Jr., & Wegbreit, B. A laboratory for the study of automating programming. **Proceedings of AFIPS Spring Joint Computer Conference**, 1972, pp. 11-21.

Cheatham, T. E., & Townley, J. A. **Symbolic Evaluation of Programs: A Look at Loop Analysis**, TR-11-76, Center for Research in Computing Technology, Harvard University, 1976.

Clark, K. and Sickel, ?. Predicate logic: A calculus for deriving programs. **IJCAI 5**, 1977, 419-420.

Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. **Structured Programming**. New York: Academic Press, 1972.

Dahl, O. J., Myhrhaug, B., & Nygaard, K. **SIMULA67 Common Base Language**, Publ. No. S-2, Norwegian Computing Centre, Oslo, 1968.

Darlington, J. **A Semantic approach to automatic program improvement**. Doctoral dissertation, University of Edinburgh, Scotland, 1972.

Darlington, J. Automatic program synthesis in second-order logic. **IJCAI 3**, 1973, 537-542.

Darlington, J. Applications of program transformation to program synthesis. In G. Huet & G. Kahn (Eds.), **Proving and Improving Programs**. Rocquencourt, France: Institut de Recherche d'Informatique et d'Automatique, July 1975. Pp. 133-144.

Darlington, J. **A Synthesis of Several Sorting Algorithms**, Research Report 23, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, July 1976.

Darlington, J., & Burstall, R. M. A System which automatically improves programs. **IJCAI 3**, 1973, 479-485.

Dershowitz, N., & Manna, Z. On automating structured programming. In G. Huet & G. Kahn (Eds.), **Proving and Improving Programs**. Rocquencourt, France: Institut de Recherche d'Informatique et d'Automatique, July 1975. Pp. 167-193.

Dershowitz, N., & Manna, Z. The evolution of programs: A system for automatic program modification. **Fourth ACM Symposium on Principles of Programming Languages**, Los Angeles, CA, January 1977.

Deutsch, B. G. The structure of task-oriented dialogs. In L. Erman (Ed.), **IEEE Symposium on Speech Recognition: Contributed Papers, IEEE Group on Acoustics, Speech, and Signal Processing**. The Institute of Electrical and Electronics Engineers, Inc., New York, April 1974. Pp. 250-254.

Deutsch B. G. **Establishing Context in Task-Oriented Dialogs**, Tech. Note 114, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA, September 1975.

Dijkstra, E. W. Guarded commands, nondeterminancy and formal derivation of programs. **CACM**, 1975, 18(8), 453-457.

Dijkstra, E. W. **A discipline of programming**. Englewood Cliffs, N.J.: Prentice-Hall, 1976.

Earley, J. Relational level data structures in programming languages. **Acta Informatica**, 1973, 2, 293-309.

Earley, J. **High-level Iterators and a Method for Automatically Designing Data Structure Representation**, Memo ERL-M425, Electronics Research Laboratory, University of California, Berkeley, 1974. (a)

Earley, J. High-level operations in automatic programming. **Proceedings of the SIGPLAN Symposium on Very High-level Languages**, March 1974. **SIGPLAN Notices**, 1974, 9(4), 34-42. (b)

Elcock, E. W., Foster, J. M., Gray, P. M. D., McGregor, J. J., & Murray, A. M. ABSET: A programming language based on sets: Motivation and examples. In B. Meltzer & D. Michie (Eds.), **Machine Intelligence 6**. Edinburgh: Edinburgh University Press, 1971. Pp. 467-492.

Feldman, J. A., Gips, J., Horning, J. J., & Reder, S. **Grammatical complexity and inference**, AI Memo 89, AI Lab, Stanford University, June 1969.

Feldman, J. A. **Towards Automatic Programming**. Preprints of the **NATO Software Engineering Conference**, Rome, Italy, October 1969.

Feldman, J. A. **Automatic Programming**, AIM-160, STAN-CS-72-255, Stanford AI Lab, Computer Science Dept., Stanford University, February 1972.

Fenichel, R. R., Weizenbaum, J., & Yochelson, J. C. A program to teach programming. **CACM**, 1970, 13(3), 141-146.

Floyd, R. W. Toward interactive design of correct programs. In C. V. Freiman (Ed.), **Foundations and Systems, Information Processing 71: Proceedings of IFIP Congress 71** (vol. 1). Amsterdam: North-Holland Publishing Co., 1972. Pp. 7-10. (Also Memo AIM-150, Report STAN-CS-71-235, AI Lab, Computer Science Dept., Stanford University, September 1971.)

Ginsparg, J. M. **Natural Language Processing in an Automatic Programming Domain**. Doctoral dissertation and Memo AIM-316, Rep. STAN-CS-78-671, AI Lab, Computer Science Dept., Stanford University, Stanford, CA, June 1978.

Goldberg, P. C. **Automatic Programming**, RC 5148, Computer Sciences Dept., Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, September 1974.

Goldberg, P. C. **The Future of Programming for Nonprogrammers**, RC 5975, Watson Research Center, IBM, Yorktown Heights, New York, May 1976.

Goldman, N., Balzer, R. M., & Wile, D. The Inference of Domain Structure from Informal Process Descriptions, **Workshop on Pattern-Directed Inference Systems**, Hawaii, May 1977. **SIGART Newsletter**, No. 63, June 1977, pp. 75-82.

Goldstein, I., & Sussman, G. J. **Some projects in automatic programming.** Working Paper 67, AI Lab, MIT, March 1974.

Goodman, R. (Ed.) **The Annual Review in Automatic Programming** (Papers of the Working Conference on Automatic Programming of Digital Computers, Brighton, April 1959). New York: Pergamon Press,1960.

Green, C. **The Application of Theorem Proving to Question-answering Systems.** Doctoral dissertation, Electrical Engineering Dept., Memo AIM-96, Report STAN-CS-69-138, AI Lab, Computer Science Dept., Stanford University, June 1969.

Green, C. Unpublished lecture surveying Automatic Proramming. Stanford University, Computer Science Dept., 1975. (a)

Green, C. **Whither automatic programming**, invited tutorial lecture. **IJCAI 4**, Tbilisi, USSR, September 1975. (b)

Green, C. An informal talk on recent progress in Automatic Programming. **Lectures on Automatic Programming and List Processing**, PIPS-R-12, Electrotechnical Laboratory, Tokyo, Japan, November 1976, pp. 1-69. (a)

Green, C. The design of the PSI program synthesis system. **Proc. 2nd International Conference on Software Engineering**, October 1976. Pp. 4-18. (b)

Green, C. The PSI Program Synthesis System, 1976. ACM '76: **Proceedings of the Annual Conference, Association for Computing Machinery**, New York, N.Y., October 1976, pp. 74-75. (c)

Green, C. A Summary of the PSI Program Synthesis System. **IJCAI 5**, 1977, 380-381.

Green, C. The PSI Program Synthesis System, 1978: An Abstract. In S. P. Ghosh, & L. Y. Liu (Eds.), **AFIPS Conference Proc.: National Computer Conf.**, 1978, 47, 673-674.

Green, C. , & Barstow, D. A hypothetical dialogue exhibiting a knowledge base for a program understanding system. In E. W. Elcock & D. Michie (Eds.), **Machine Intelligence 8: Machine Representations of Knowledge.** New York: Halsted Press, John Wiley & Sons, 1977. Pp. 335-359. (a)

Green, C. , & Barstow, D. **Some rules for the automatic synthesis of programs. IJCAI 4**, 1975, 232-239.

Green, C., et al. **Progress Report on Knowlege Based Programming.** Systems Control, Inc., Computer Science Division, Palo Alto, CA, September 1978.

Green, C., & Barstow, D. **On Program Synthesis Knowledge**, Memo AIM-306, Report STAN-CS-77-639, AI Lab, Computer Science Dept., Stanford University, Stanford, CA, November 1977. (b)

Green, C., & Barstow, D. **On program synthesis knowledge. Artificial Intelligence**, 1978, 10(3), 241-279.

Green, C., Waldinger, R., Barstow, D., Elschlager, R., Lenat, D., McCune, B., Shaw, D., & Steinberg, L. Progress Report on Program Understanding Systems, Memo AIM-240, AI Lab, Stanford, CA, August 1974.

Gries, D. Programming by Induction, TR 71-106, Computer Science Dept., Cornell University, September 1971.

Guttag, J. V., Horowitz, E., & Musser, D. R. Abstract Data Types and Software Validation, Tech. Report ISI-RR-76-48, Information Sciences Institute, Marina del Rey, CA, August 1976.

Hammer, M. Automatic Programming: An Assessment. Unpublished paper, MIT Lab for Computer Science, Cambridge, Mass., December 1977.

Hammer, M., & Ruth, G. Automating the Software Development Process. In P. Wegner (Ed.), Research Directions in Software Technology. Cambridge: MIT Press, 1979. Pp. 767-792.

Hammer, M., Howe, W. G., Kruskal, V. J., & Wladawsky, I. A Very High-Level Programming Language for Data Processing Applications, RC 5583, Computer Sciences Dept., Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, August 1975.

Hardy, S. Synthesis of LISP functions from examples. IJCAI 4, 1975, 240-245.

Heidorn, G. E. The End of the User Programmer? The Software Revolution, Infotech State of the Art Conf., Copenhagen, Denmark, October 1977. (To appear in Future Programming, Infotech, England, 1979.)

Heidorn, G. E. Natural Language Inputs to a Simulation Programming System, Report 55hd72101A, Naval Postgraduate School, Monterey, CA, October 1972.

Heidorn, G. E. English as a very high level language for simulation programming. IBM Research 4536, September 1973.

Heidorn, G. E. English as a very high level language for simulation programming, Proceedings Symposium on Very High Level Languages. SIGPLAN Notices, 1974, 9(4), 91-100.

Heidorn, G. E. Augmented Phrase Structure Grammars. In B. L. Nash-Webber & R. C. Schank (Eds.), Theoretical Issues in Natural Language Processing. Association for Computational Linguistics, June 1975. Pp. 1-5. (a)

Heidorn, G. E. Simulation programming through natural language dialogue. Amsterdam: North-Holland Studies in the Management Sciences, 1975. (b)

Heidorn, G. E. Simulation programming through natural language dialogue. In M. A. Geisler (Ed.), TIMS Studies in the Management Sciences, Logistics (vol. 1). Amsterdam: North Holland, 1975. Pp. 71-85. (c)

Heidorn, G. E. Automatic programming through natural language dialogue: A survey. IBM Journal of Research and Development, 1976, 20(4), 302-313.

Henderson, P., & Morris, J. H., Jr. A lazy evaluator. **Third ACM Symposium on Principles of Programming Languages**, Atlanta, GA, January 1976. Pp. 95-103.

Hewitt, C. **Teaching Procedures in Humans and Robots**, Memo 208, AI Lab, Massachusetts Institute of Technology, April 1970.

Hewitt, C. **Viewing Control Structures as Patterns of Passing Messages**, Working paper 92 (rev. ed.), AI Lab, Massachusetts Institute of Technology, April 1976.

Hewitt, C., & Smith, B. C. Towards a programming apprentice. **IEEE Transactions: Software Engineering**, 1975, 1(1), 26-45.

Hill, I. D. Wouldn't it be nice if we could write computer programs in ordinary English--or would it? **Computer Bulletin**, 1972, 16(6), 306-312.

Hobbs, J. R. **From Well Written Algorithm Descriptions into Code**, Research Rep. 77-1, Dept. of Computer Science, City College, City University of New York, July 1977.

Kant, E. The selection of efficient implementations for a high level language, **Proceedings of Symposium on Artificial Intelligence and Programming Languages**. **SIGPLAN Notices**, 12(8); **SIGART Newsletter**, No. 64, August 1977, pp. 140-146.

Kant, E. Efficiency Estimation: Controlling Search in Program Synthesis. In S. P. Ghosh & L. Y. Leonard (Eds.), **AFIPS Conf. Proc.: National Computer Conf.**, 1978, 47, 703.

Kant, E. **Efficiency Considerations in Program Synthesis: A Knowledge-based Approach**. Doctoral dissertation, Stanford University, Computer Science Dept., 1979.

Kowalski, R. **Predicate Logic as a Programming Language**, Information Processing, North Holland, Amsterdam, 1977.

Lenat, D. B. Synthesis of large programs from specific dialogues. In G. Huet & G. Kahn (Eds.), **Proving and Improving Programs**. Rocquencourt, France: Institut de Recherche d'Informatique et d'Automatique, July 1975. Pp. 225-241.

Liskov, B. H., Snyder, A., Atkinson, R., & Schaffert, C. Abstraction Mechanisms in CLU. **CACM**, 1977, 20(8), 564-576.

Lomet, D. B. **Data Flow Analysis in the Presence of Procedure Calls**, RC 5728, Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, November 1975.

Long, W. J. **A Program Writer**. Doctoral dissertation, TR-187, LCS, Massachusetts Institute of Technology, November 1977.

Low, J. P. **Automatic coding: Choice of data structures**, Stanford AI Memo AIM-242, Stanford University, August 1974.

Low, J. R. **Automatic Coding: Choice of Data Structures**, ISR16, Birkhauser Verlag, 1976. (a)

Low, J. R. **Automatic Data Structure Selection: An Example and Overview**, TR-14, Computer Science Dept., University of Rochester, September 1976. (b)

Low, J. R. Automatic data structure selection: An example and overview. **CACM**, 1978, **5**, 21-25.

Manna, Z., & Waldinger, R. DEDALUS--The DEDuctive ALgorithm UR-Synthesizer. **National Computer Conference**, Anaheim, CA, June 1978. Pp. 683-690.

Manna, Z., & Waldinger, R. **Synthesis: Dreams Programs**, Memo AIM, AI Lab, Stanford, CA, November 1977.

Manna, Z., & Waldinger, R. J. Toward automatic program synthesis. **Communications of the ACM**, 1971, **14**(3), 151-165.

Manna, Z., & Waldinger, R. Knowledge and reasoning in program synthesis. **Artificial Intelligence**, 1975, **6**(2), 175-208.

Martin, W. A. **OWL Notes: A System for Building Expert Problem Solving Systems Involving Verbal Reasoning**, M.I.T. Project MAC, 1974.

McCune, B. P. The PSI program model builder: Synthesis of very high-level programs. **Proceedings of the Symposium on Artificial Intelligence and Programming Languages**. **SIGPLAN Notices**, **12**(8), 130-139; **SIGART Newsletter**, No. 64, August 1977, 130-139.

McCune, B. P. **Building Program Models Incrementally from Informal Descriptions**. Doctoral dissertation, AI Lab Memo, Computer Science Dept., Stanford University, in press.

Michie, D. Memo functions and machine learning. **Nature**, 1968, **218**(No. 5136), 19-22.

Miller, L. A., & Becker, C. A. **Programming in Natural English**, Research Report RC 5137, Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, November 1974.

Mitchell, J. G **The Design and Construction of Flexible and Efficient Interactive Programming Systems**. Doctoral dissertation, Dept. of Computer Science, Carnegie-Mellon University, June 1970.

Morgenstern, M. **Automatied Design and Optimization of Information Processing Systems**. Doctoral dissertation, MIT, 1976.

Persson, S. **Some Sequence Extrapolating Programs: A Study of Representation and Modeling in Inquiring Systems**. Doctoral dissertation, School of Business Administration, University of California, Berkeley; Memo AIM-46, Report STAN-CS-66-50, AI Lab, Computer Science Dept., Stanford University, September 1966.

Petry, F. E., & Biermann, A. W. Reconstruction of algorithms from memory snapshots of their execution. **ACM '76: Proceedings of the Annual Conference**, Association for Computing Machinery, New York, October 1976, pp. 530-534.

Phillips, J. V. Program Inference from Traces Using Multiple Knowledge Sources. **IJCAI 5**, 1977, p. 812.

Pratt, V. R. **The Competence/Performance Dichotomy in Programming.** TM-400, AI Lab, Massachusetts Institute of Technology, January 1977.

Project MAC, Automatic composition of functions from modules (Section III E.1). **Project MAC Progress Report X**, Massachusetts Institute of Technology, July 1972-July 1973.

Reisser, J. F. (Ed.) **SAIL**, Stanford AI Memo No. 289, August 1976.

Rich, C. **A Library of Programming Plans with Applications to Automated Analysis, Synthesis and Verification of Programs.** Doctoral dissertation, MIT, Cambridge, MA, 1979.

Rich, C., & Shrobe, H. E. **Initial Report on a LISP Programmer's Apprentice**, TR-354, AI Lab, Massachusetts Institute of Technology, December 1976.

Rich, C., & Shrobe, H. Initial Report on a LISP Programmer's Apprentice. **IEEE Trans. on Soft. Eng.**, 1978, **4**(6), 456-467.

Rivest, R. L. **Two-Dimensional Programming Languages.** Dept. of Electrical Engineering and Computer Science Dept., MIT, April 1975.

Rosen, B. K. **Data Flow Analysis for Procedural Languages**, RC 5948, Computer Sciences Dept., Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, April 1976.

Rosen, B. K. Applications of high-level control flow. **Fourth ACM Symposium on Principles of Programming Languages**, Los Angeles, CA, January 1977.

Rovner, P. D. **Automatic Representation Selection for Associate Data Structures**, TR-10, Computer Science Dept., University of Rochester, September 1976.

Ruth, G. **Analysis of Algorithm Implementations.** Doctoral dissertation, TR-130, Project MAC, Massachusetts Institute of Technology, May 1974.

Ruth, G. Automatic Design of Data Processing Systems, **Proc. of the Third ACM Symposium on Principles of Programming Languages**, Atlanta, Georgia, 1976 (also MIT Comp. Sci. TR TM-070). (a)

Ruth, G. Intelligent program analysis. **AI**, 1976, **7**, 65-85. (b)

Ruth, G. Protosystem I: An Automatic Programming System Prototype, **Proc. of the National Computer Conf.**, Anaheim, CA, 1978. **AFIPS**, 1978, **47**, 675-681.

Ruth, G. Automating the Software Development Process. In P. Wegner (Ed.), **Research Directions in Software Technology.** Cambridge: MIT Press, 1979.

Sacerdoti, E. D. **A Structure for Plans and Behavior.** New York: Elsevier, 1977.

Schwartz, J. T. **On Programming: An Interim Report on the SETL Project** (rev. ed.). Computer Science Dept., Courant Institute of Mathematical Sciences, New York University, June 1975.

Shaw, D., Swartout, W., & Green, C. Inferring LISP programs from examples. **IJCAI 4**, 1975, 260-267.

Shrobe, H. E. **Reasoning and Logic for Complex Program Understanding.** Doctoral dissertation, MIT, Cambridge, MA, August 1978.

Sibel, W., Furbach, U., & Schreiber, J. F. Strategies for the synthesis of algorithms. **Informatik-Fadbendik**, 1978, **5**, 97-109.

Siklossy, L. The synthesis of programs from their properties, and the insane heuristic. **Proceedings of the Third Texas Conference on Computing Systems**, Austin, TX, 1974, pp. 5-2-1 - 5-2-5.

Siklossy, L., & Sykes, D. Automatic program synthesis from example problems. **IJCAI 4**, 1975, 268-273.

Simon, H. A. Experiments with a heuristic compiler. **JACM**, 1963, **10**(4), 493-503.

Simon, H. A. The heuristic compiler. In H. A. Simon & L. Siklossy (Eds.), **Representation and Meaning**. Englewood Cliffs, N. J.: Prentice-Hall, 1972. Pp. 9-43.

Summers, P. D. A Methodology for LISP Program Construction from Examples. **JACM**, 1977, **24**(1), 161-175.

Sussman, G. J. **A Computer Model of Skill Acquisition**. New York: American Elsevier, 1975.

Szolovits, P., Hawkinson, L. B., & Martin, W. A. **An Overview of OWL, A Language for Knowledge Representation**, TM-86, LCS, Massachusetts Institute of Technology, June 1977.

Teitelman, W. **PILOT: A Step Toward Man-Computer Symbiosis**. Doctoral dissertation, MAC-TR-32, Project MAC, Massachusetts Institute of Technology, September 1966.

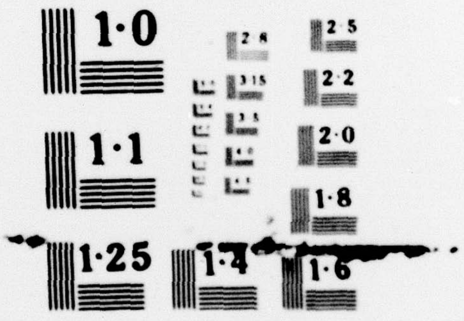
Teitelman, W. Toward a programming laboratory. **IJCAI 1**, 1969, 1-8.

Teitelman, W. Automated programming--The programmer's assistant. **Proceedings Fall Joint Computer Conference** (Vol. 41), December 1972, pp. 917-921.

Teitelman, W. **Interlisp Reference Manual**. Xerox Corp., Palo Alto, CA, 1974.

Teitelman, W. **A Display Oriented Programmer's Assistant**, CSL 77-3, Palo Alto Research Center, Xerox Corp., Palo Alto, CA, March 1977.

Teitelman, W., et al. **INTERLISP Reference Manual**. Xerox PARC, Palo Alto, CA, October 1978.


1·0
2·8
2·5
3·15
2·2
3·5
2·0
1·1
1·8
1·25
1·4
1·6

Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., & Koster, C. H. A. Report on the algorithmic Language ALGOL68. Numerische Mathematik, 1969, 14, 79-218.

Waldinger, R. Constructing Program Automatically Using Theorem Proving. Doctoral dissertation, Carnegie-Mellon University, Pittsburgh, Penn., 1969.

Waldinger, R. Achieving several goals simultaneously. In E. W. Elcock & D. Michie (Eds.), Machine Intelligence 8: Machine Representations of Knowledge. New York: Halsted Press, John Wiley & Sons, 1977. Pp. 94-136.

Waldinger, R. , & Lee, R. PROW: A step toward automatic program writing. IJCAI 1, 1969, 241-252.

Waldinger, R. , & Levitt, K. N. Reasoning about programs. Artificial Intelligence, 1974, 5(3), 235-316.

Warren, D. H. D. WARPLAN: A System for Generating Plans, Memo No. 76, Dept. of Computational Logic, School of Artificial Intelligence, University of Edinburgh, Scotland, June 1974.

Warren, D. H. D. Generating conditional plans and programs. Proceedings of the Conference on Artificial Intelligence and Simulation of Behavior, Edinburgh, Scotland, July 1976, pp. 344-354.

Warren, D. H. D. Implementing PROLOG: Compiling Predicate Logic Programs (vols. 1-2), Research Reports 39-40, Dept. of AI, University of Edinburgh, Scotland, May 1977.

Warren, H. S., Jr. Data Types and Structures for a Set Theoretic Programming Language, RC 5567, Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, August 1975.

Waterman, D. A. Generalization Learning Techniques for Automating the Learning of Heuristics. Artificial Intelligence, 1970, 1, 121-170.

Waters, R. C. A System for Understanding Mathematical FORTRAN Programs, MIT-AIM-168, MIT, Cambridge, MA, August 1976.

Waters, R. C. Automatic Analysis of the Logical Structure of Programs, MIT-AI-TR-492, December 1978 (based on doctoral dissertation, A Method for Automatically Analyzing the Logical Structure of Programs, August 1978).

Waters, R. C. A Method for Analyzing Loop Programs. To appear in IEEE Trans. on Soft. Eng., 1979.

Wegbreit, B. Studies in Extensible Programming Languages. Doctoral dissertation, Center for Research in Computing Technology, Harvard University, January 1972.

Wegbreit, B. Goal-directed program transformation, CSL-75-8, Xerox PARC, Palo Alto, CA, September 1975. (a)

Wegbreit, B. Mechanical program analysis. CACM, 1975, 9(18), 528-539. (b)

Wilber, B. M. A QLISP Reference Manual, AI Center Tech. Report 118, SRI International, Inc., Menlo Park, CA, March 1976.

Winograd, T. Five Lectures on Artificial Intelligence, Stanford AIM-246, CS459, Computer Science Dept., Stanford University, September 1974.

Winograd, T. Breaking the complexity barrier again. SIGPLAN Notices, 1975, 10(1), 13-30.

Winston, P. H. Learning structural descriptions from examples. In P. Winston (Ed.), The Psychology of Computer Vision. New York: McGraw-Hill, 1975.

Wirth, N. The programming language PASCAL. Acta Informatica, 1971, 1, 35-63.

Zilles, S. Abstract Specification for Data Types. IBM Research Laboratory, San Jose, CA, 1975.

Zimmerman, L. L. On-line program debugging: A graphic approach. Computers and Automation, 1967, 16(11), 30-34.